INFORMATION AND COMPUTATION 93, 114–171 (1991)

A Model of Concurrency with Fair Merge and Full Recursion*

YIANNIS N. MOSCHOVAKIS†

Department of Mathematics, University of California at Los Angeles, Los Angeles, California 90024

DEDICATED TO THE MEMORY OF DAVID PARK

We introduce in this paper a model for interaction and asynchronous, concurrent communication, where each agent's *perception of the system* is represented by a game of interaction. The model combines (strict) fair merge with full recursion, and the main mathematical results provide evidence for the robustness and naturalness of a novel interpretation of recursive definitions of non-deterministic processes. Our conceptual approach is closest to Park's, whose ideas are the starting points for this work. © 1991 Academic Press, Inc.

1. INTRODUCTION

Smyth (1978) begins with the observation that "to apply the methods of fix-point semantics [to non-deterministic programs], we should find some way to construe the powerset of a domain as itself a domain, with a suitable ordering." There is a difficulty with such a program, however, if among the non-deterministic constructs of our language we want to include the *fair merge*, as it leads to complex sets which cannot be embedded naturally in a domain. In the abstract of his first paper on powerdomains, Plotkin (1976) comments that "the main deficiency is the lack of a convincing treatment of the fair parallel construct."

One of the two main aims of this paper is to define natural and useful denotational semantics for non-deterministic languages which include both full recursion and the fair merge operation. We will not do this with a powerdomain construction, but with a novel interpretation of recursive definitions which makes it possible "to apply the methods of fix-point semantics" in a precise, technical sense, although it is not "least-fix-point" with respect to any natural ordering. Thus our semantics go beyong

* A preliminary version of this paper (Moschovakis (1989b)) appeared in the *Proceedings of the 1989 LICS Conference* with the title "A Game-Theoretic Modeling of Concurrency." It was convenient to include in this final version some relevant material of the later announcement (Moschovakis (1990)) which fits in directly with the results here.

† During the preparation of this paper the author was partially supported by an NSF Grant.

0890-5401/91 $3.00

Copyright © 1991 by Academic Press, Inc.
All rights of reproduction in any form reserved.

domain theory and we shall be forced to justify that they are "convincing" *ab initio.*

The second main aim of the paper is to introduce a conceptual framework for studying non-deterministic interaction, in which the implementations of a concurrent program or *agent* X are taken to be *strategies* in a natural two-person game: X is trying to control certain aspects of the overall behavior of the system, seemingly against the wishes of the other agents who are operating in the same environment and appear (to X) to be acting as a single, powerful "opponent," bent on frustrating his efforts. This is a natural view of interactive programming and it yields simple and direct game-theoretic definitions of the most basic operations on processes. More significantly, this game imagery also yields a novel and intuitively appealing solution to the problem of *synchronization* of conflicting, asynchronous, concurrently executed programs which must be resolved in any modeling of concurrency.

It is possible to separate these two strands of the paper, but the results of each reinforce the other and together they give a coherent theory of interaction and concurrency. We begin with a section of preliminaries which makes the paper accessible to the non-expert—the expert should skip through this quickly, just to pick up the (mostly standard) notation and terminology. In Sections 3 and 4 we describe the game-theoretic modeling of processes, and in Section 5 we apply it to the theory of non-deterministic networks. In Section 6 we set the stage for the more technical development which follows by formulating precise, minimum conditions which should be satisfied by any reasonable interpretation of recursion in the presence of the merge construct. Sections 7–9 develop the theory of non-deterministic recursion, with the proof of the main, technical result relegated to the Appendix, Section 10.

To illustrate the notions, we will refer to the following three standard, well known examples which involve the basic puzzles of concurrency modeling.

1.1. *Park's example* (Park, 1980). Let

$$P \equiv X := 0 \ \textit{next} \ Y := 0 \ \textit{next par} \ (X := 1, \textit{while} \ (X = 0) \ Y := Y + 1),$$

where $par(E, M)$ denotes the *strict, fair merge* of the processes E, M. Park argues (essentially) that from a correct understanding of this definition we should be able to make precise and prove that *in an environment where no other process can write to the variable X, P will terminate in some indeterminate state* ($X = 1$, $Y = n$). The indeterminacy of the final state expresses the "unbounded non-determinism" deliberately put in this definition: i.e., in arguing about P we should be able to assume that it terminates and assigns 1 to X and some integer to Y, but nothing more.

1.2. *Dijkstra's dining philosophers* (Dijkstra, 1976). Five philosophers D_0 ..., D_4 (named by the integers mod 5) spend their lives alternatively thinking and eating. Each philosopher can commence thinking whenever he wants, but because of a peculiar arrangement in the dining room, D_i cannot share the table with either of his adjacent colleagues D_{i-1}, D_{i+1}. If we describe the life of each D_i by the recursion

$$D_i \equiv eat_i \; next \; think_i \; next \; D_i, \tag{1}$$

then the communal life of the system could be specified by

$$D \equiv par(D_0, ..., D_4). \tag{2}$$

A correct modeling of this system should assign to the process D *all deadlock-free and fair behaviors* allowed by the given constraint, i.e., all possible "scheduling algorithms" which make it possible for each D_i to continue indefinitely alternating eating and thinking.

Discussions of Dijkstra's example sometimes allow interpretations which permit deadlock and almost always include the non-fair behaviors in D, where some D_i may starve after a certain stage. Perhaps this is because in his classic monograph Dijkstra (1976) rejects unbounded non-determinism—and hence fair merge. Here we side with Park (1980, 1983) on the issue of fair merge and we accept the natural (in our view), fair interpretation of (2). For our purposes, the most interesting aspect of Dijkstra's example is that it makes it clear just how complex the "compatibility relations" can be among acts which we might want to have executed concurrently—and correspondingly, how complex the appropriate "non-deterministic merging" operation may have to be.

1.3. *The merge anomalies.* Suppose we have a distinct act n for each integer and let $S(x)$ be an operation on processes such that the process $S(x)$ copies exactly the process x except that it replaces execution of each n by $n+1$. Let M be the non-deterministic process defined by the recursion

$$M = par(5, S(M)). \tag{3}$$

It is clear that among the behaviors of M is the natural one which executes in succession 5, 6, ... It follows that M must also exhibit a behavior which begins by executing 6, but that is paradoxical: execution of 6 must be "driven" by an execution of 5, so how can it precede it?

This is perhaps the simplest of several examples of *merge anomalies* discussed extensively in the literature, cf. Keller (1978), Brock and Ackerman (1981), Park (1980, 1983), Broy (1986), and Oles (1987). It comes from interpreting (3) as the recursive definition of a *non-deterministic network* and it has led to suggestions that processes defined by recursion need not

satisfy their defining equations, cf. Broy (1986). In our modeling, recursion will keep its fundamental fixed point property, but non-deterministic networks will be understood differently.

2. Preliminaries

2.1. By a *simultaneous* (or mutual) *recursive definition* we mean the assignment of solutions $\bar{x}_1, ..., \bar{x}_n$ to a system of equations

$$\begin{aligned} x_1 &= f_1(x_1, ..., x_n) \\ &\vdots \\ x_n &= f_n(x_1, ..., x_n) \end{aligned} \tag{4}$$

for given functions $f_i: X_1 \times \cdots \times X_n \to X_i$ $(i = 1, ..., n)$. In the classical case of least fixed point recursion, each X_i is a *dcpo*, i.e. it comes equipped with a partial ordering $\leqslant$ in which directed sets have least upper bounds. In particular each X_i has a least element, $\min_i = \sup \varnothing$. It follows that the product partial ordering

$$(x_1, ..., x_n) \leqslant (x'_1, ..., x'_n) \Leftrightarrow x_1 \leqslant_1 x'_1 \;\&\cdots\&\; x_n \leqslant_n x'_n$$

on $X = X_1 \times \cdots \times X_n$ is also a dcpo, and we are further given that each function f_i is *monotone* and (countably) *continuous* as follows:

$$\vec{x} \leqslant \vec{x}' \Rightarrow f_i(\vec{x}) \leqslant_i f_i(\vec{x}').$$

$$\vec{x}_0 \leqslant \vec{x}_1 \leqslant \cdots \Rightarrow f_i\left(\bigcup_i \vec{x}_i\right) = \bigcup_i f_i(\vec{x}_i).$$

Under these hypotheses, we set for $i = 1, ..., n$ and $k = 0, 1, ...$

$$\begin{aligned} \bar{x}_i^{(0)} &= f_i(\min_1, ..., \min_n), \\ \bar{x}_i^{(k+1)} &= f_i(\bar{x}_1^{(k)}, ..., \bar{x}_n^{(k)}), \\ \bar{x}_i &= \bigcup_k \bar{x}_i^{(k)}, \end{aligned}$$

and we verify easily that the tuple $\bar{x}_1, ..., \bar{x}_n$ satisfies the system (4) and that it is in fact *the least solution*, in the strong sense that for all $x'_1, ..., x'_n$,

$$(\text{for } i = 1, ..., n)[f_i(x'_1, ..., x'_n) \leqslant_i x'_i] \Rightarrow (\text{for } i = 1, ..., n)[\bar{x}_i \leqslant_i x'_i].$$

We call the objects $\bar{x}_i^{(k)}$ *the stages* of the recursion and we will sometimes give proofs *by induction on the stages*.

More generally, we may seek functions $\bar{x}_i: Y \to X_i$, $i = 1, ..., n$ which satisfy a system with parameters

$$\begin{aligned} x_1 &= f_1(x_1, ..., x_n, y) \\ &\vdots \\ x_n &= f_n(x_1, ..., x_n, y). \end{aligned} \tag{5}$$

If the given functions are monotone and continuous on given dcpo's, then the same method yields (pointwise) least solutions which are themselves monotone and continuous.

Here we will need to solve systems like (4) and (5) where the variables model *non-deterministic processes* and vary over collections of sets which do not carry natural, nice partial orderings, so that the *process functions* in the system cannot be viewed usefully as "monotone and continuous." The main mathematical contribution of the paper consists in developing a theory of simultaneous recursive definition for such systems and comparing it with least fixed point recursion. As a tool for this comparison we will use a simple language $\mathscr{L}$ whose main construct is recursion and for which we will define several natural semantics, including the "process semantics" which are our main concern.

2.2. *The language $\mathscr{L}$.* A *signature* τ is any set of *function symbols*, each with an assigned non-negative integer, its *arity*. The expressions of $\mathscr{L} = \mathscr{L}(\tau)$ are defined by the induction

$$E ::\equiv x \mid f(E_1, ..., E_n) \mid \mathtt{rec}(x_1, ..., x_n)[E_0, E_1, ..., E_n],$$

where x is any variable (from a fixed, infinite set), f is any function symbol of arity n, and a more familiar notation for the recursion construct is

$$E_0 \,\mathtt{where}\, [x_1 = E_1, ..., x_n = E_n]. \tag{6}$$

The variables $x_1, ..., x_n$ are bound in the expression (6) and its intended value is the value of E_0 when the variables are assigned the solutions of the system

$$x_i = E_i \qquad (i = 1, ..., n),$$

however these solutions are obtained.

2.3. *Least-fixed-point semantics.* The simplest interpretation of $\mathscr{L}$ is on structures of the form $\mathcal{O} = (\leqslant, \mathscr{F})$, where $\leqslant$ is a dcpo and $\mathscr{F}$ is a map which assigns to each function symbol f a continuous, monotone function $\mathscr{F}(f)$ of the correct arity. We let the variables range over the field of $\leqslant$ and we associate with each expression E and each list of n variables $\vec{x}$

which includes all the free variables of E a continuous, monotone n-ary function f_E in the obvious way, taking least-fixed-points to interpret the rec construct. We call two $\mathscr{L}$ expressions E, M least-fixed-point equivalent if they are assigned the same function $f_E = f_M$, on all structures $\mathcal{O}$ and for all $\vec{x}$.

2.4. A *state structure* is a tuple of the form

$$\mathscr{S} = (\mathsf{States}, \iota, \mathsf{Acts}, skip, exec), \tag{7}$$

where the following hold:

1. States is some set of *states* which contains the *initial state* ι.
2. Acts is some set of *atomic acts* which includes the no-act act *skip*.
3. *exec*: States $\times$ Acts $\to$ Acts is a binary operation of act execution, so that each act a induces a (total) transition function

$$s \mapsto sa = exec(s, a)$$

on the states. We assume that $s\, skip = s$, for every state s.

A state structure is *trivial* if it has only the one state ι.

A *history* is a finite sequence $h = a_1 a_2 \cdots a_n$ of acts which acts on States in the obvious way,

$$sh = (\cdots (sa_1) a_2) \cdots) a_n.$$

In one extreme case states may be identified with histories. More typically the state is a *store*, for (a toy) example

$$s = (x, y, (z_0, ..., z_n), (b_0, ..., b_m)),$$

in this case keeping track of the integer contents of two integer variables X, Y, a stack Z and a buffer B, presumably to be used for sending messages along some channel. The obvious acts to put in such a structure would include assignments $X := w$, $Y := w$ to the variables $push^Z_w$ and pop^Z on the stack and the sending and receiving of messages using the buffer, $send^B_w$, $receive^B$, which alter the state in the obvious way. There may also be other acts which do not change the state, e.g. $write_X$, which (presumably) prints the current value x of the variable X on some device; we do not assume that distinct acts induce distinct transition functions on the state.

It is quite obvious how to interpret the expressions of $\mathscr{L}$ on such structures, letting the variables vary over "procedures" (denotations of programs or program fragments) and interpreting the function symbols by given "program transformations." We will outline the precise definitions,

partly to set notation and because they will form a basis for our generalizations later.

2.5. As usual, a *convergent stream* from a set U is a finite sequence of the form $(u_1, u_2, ..., u_n, 1)$ where $n > 0$, each u_i is in U, and 1 is a special *termination indicator*; a *divergent* (or partial) *stream* from U is any finite (possibly empty) or infinite sequence of members of U. For each U, the set Streams(U) of streams from U is naturally partially ordered by the "initial segment" relation, with which it is a dcpo; convergent and infinite streams are maximal elements. *Concatenation* of streams is defined so that if u is divergent then $u * v = u$ and if u is convergent, then $u * v$ is the concatenation of u and v as sequences, with the terminator 1 removed from u.

A stream of acts assigns in the obvious way to each state a stream of states, and a convergent stream of acts further induces a transition function on the states,

$$(s, (a_1, ..., a_n, 1)) \mapsto s(a_1, ..., a_n, 1) = s a_1 a_2 \cdots a_n.$$

2.6. A *procedure* is a function on states to streams of acts,

$$\alpha: \text{States} \to \text{Streams(Acts)}$$

We take procedures to be the natural denotations of non-interactive, deterministic programs; e.g., a program which prints out the sequence of primes or another which reads from the state the value x of some integer variable and eventually sends along a channel the value $f(x)$ of some f at x. In a trivial structure the state dependence is of no consequence and a procedure is just a stream of acts. The set Π of procedures is a dcpo with the natural partial ordering

$$\alpha \leqslant \beta \Leftrightarrow (\forall s)[\alpha(s) \text{ is an initial segment of } \beta(s)],$$

whose minimum is the constant procedure $\lambda(s)\varnothing$.

2.7. A *procedure function* of n arguments is any monotone and continuous function

$$\phi: \Pi^n \to \Pi.$$

Examples of procedure functions are the constant (0-ary and state independent) *act executions*

$$a = do(a) = \lambda(s)(a, 1),$$

one for each $a \in$ Acts, the *sequential execution*

$$\alpha \; next \; \beta = \lambda(s)[\alpha(s) * \beta(s\alpha(s))], \tag{8}$$

and the *conditionals*

$$cond_R(\alpha, \beta) = \lambda(s)\ [\textit{if } R(s) \textit{ then } \alpha(s) \textit{ else } \beta(s)],$$

one for each relation R on the states. Note that in (8) the state $s\alpha(s)$ is defined in the case $\alpha(s)$ is convergent, when the concatenation function needs it. In a trivial structure procedure functions are just monotone, continuous functions on streams.

2.8. A *procedure structure* is a pair $\mathscr{A} = (\mathscr{S}, \mathscr{F})$, where $\mathscr{S}$ is a state structure and $\mathscr{F}$ assigns an n-ary procedure function $\mathscr{F}(f)$ to each function symbol of arity n. To interpret the expressions of $\mathscr{L}$ in a procedure structure, we let the variables vary over procedures and we interpret the recursion construct by the taking of simultaneous least-fixed-points. In this way, for each procedure structure $\mathscr{A}$, each expression E, and each n-tuple of variables $\vec{x}$ which includes all the free variables of E, we obtain an n-ary procedure function

$$\phi_E = procedure(\mathscr{A}, \vec{x})E,$$

the *procedure denotation of E relative to* $\vec{x}$. If E is a *closed expression* (with no free variables), then its procedure denotation ϕ_E relative to the empty list is a function with no arguments, i.e., just a procedure. Two expressions E, M are *procedure equivalent* if they have the same procedure denotation $\phi_E = \phi_M$ on every procedure structure and for every $\vec{x}$. Since procedure semantics is a special case of least-fixed-point semantics, it follows that *least-fixed-point equivalent expressions are also procedure equivalent.*

With this simple denotational semantics $\mathscr{L}$ is a bare-bones command language. Because it allows full recursion, however, it is not entirely trivial; on a reasonable structure $\mathscr{A}$ with a usable conditional, act and sequential execution, and a rich enough state to allow for some value passing, it can be a "complete" programming language.

3. The Game of Interaction

The procedure semantics of $\mathscr{L}$ just described assume that its expressions (programs) define non-interactive algorithms, intended for execution in an environment which leaves them alone: a procedure reads the state just once (most likely to get input) and then proceeds to execute a stream of acts free of external interference. This is quite unrealistic, of course, even if only because execution of any program may be interrupted by a deliberate act of the operating system—or by a disaster! Especially for interactive programs, it is more profitable to use for denotations "extended proce-

dures" which (at least in theory) consult the state before each act execution, not just once in the beginning.

We can also view the expressions of $\mathscr{L}$ as specifications for the interactive behaviors of "computing agents" operating in a shared environment. Such a computing agent may encounter an unexpected, new state each time he interacts with his environment, because of what all the other agents did "while he was not looking." *From his perspective,* the agent may well perceive the situation as a two-person game in which his (apparently single) "opponent" takes turns with him acting on the state; and we may model the agent's behavior by the *strategy* that he is following in that game. We now turn this intuitive imagery into a precise "interactive semantics" for $\mathscr{L}$.[1]

3.1. *The game.* With each state structure $\mathscr{S}$ as in (7) we associate the two-person *game of interaction* $G = G(\mathscr{S})$ where player II represents some computing agent or program fragment and player I represents everybody else—*the world.* A *run* of G is played in stages, with I and II exchanging moves at the nth stage so that they may alter *the state* g_n initially set by $g_{-1} = \iota$. At stage n, I plays first some state s_n which must be *accessible* from g_{n-1}, i.e., such that $s_n = g_{n-1}h$ for some history h; II then responds with a pair (a_n, w_n) of an act a_n and an *indicator* $w_n = \partial$ or $w_n = 1$ and the next state is set $g_n = s_n a_n$; if $w_n = 1$, the run ends, otherwise we proceed to the next stage $n + 1$.

We read drawings like

$$\begin{array}{llllll} \text{I} & s_0 & s_1 & s_2 & s_3 & \\ & & & & & \cdots \\ \text{II} & (a_0, w_0) & (a_1, w_1) & (a_2, w_2) & (a_3, w_3) & \\ \text{State: } \iota & s_0 a_0 & s_1 a_1 & s_2 a_2 & s_3 a_3 & \cdots \end{array} \tag{9}$$

from left-to-right and top-to-bottom; i.e., the indicated moves produce the stream

$$\rho = s_0, (a_0, w_0), s_1, (a_1, w_1), s_2, (a_2, w_2), \ldots, \tag{10}$$

which is a typical *run* of G. The game is "of perfect information"; i.e., each player knows all the moves already made when it is his turn to move. We think of the act a_n as the main part of II's move, with the indicator $w_n = \partial$ signifying that II wants another turn while $w_n = 1$ announces II's exit from the game.

[1] Similar modelings of interactive and concurrent systems in terms of games have also been presented recently in Abadi, Lamport, and Wolper (1989) and Pnueli and Rosner (1989), and (apparently) they have antecedents in older work of Lamport.

A *payoff* for the game G is any set W of runs as in (10), and we say that II *wins a run* ρ if $\rho \in W$. Such payoff sets model *specifications* for interactive programs: an *implementation* then of the specification W would be any *winning strategy* for II, any method for II to play which insures that the resulting run satisfies the given specification. On this picture, interactive programming amounts to defining winning strategies for the game G, relative to given specifications. Here, however, we are also interested in the *losing* strategies for II, since we want to model *all* interactive behaviors, not only the successful ones. So, peculiar as it may sound, we will not need to mention payoff sets again in the sequel.

To deal with finite sequences of states and acts we will adopt the notation

$$\bar{u}(n) = (u_0, ..., u_n).$$

A *world strategy* is a function

$$\omega: \mathsf{Acts}^* \to \mathsf{States}$$

on finite equances of acts such that $\omega(\varnothing)$ is accessible from the initial state ι and for each $\bar{a}(n)$ $(n > 0)$, $\omega(\bar{a}(n))$ is accessible from $\omega(\bar{a}(n-1))a_{n-1}$. Of special interest is the "inert" world strategy

$$\tilde{\omega}(\varnothing) = \iota, \qquad \tilde{\omega}(a_0, ..., a_n) = \iota a_0 a_1 \cdots a_n \tag{11}$$

which represents "no action" by other agents.

A *partial* (agent) *strategy* is any partial function

$$\sigma: \mathsf{States}^* \to \mathsf{Acts} \times \{\partial, 1\}$$

on non-empty sequences of states. (Ultimately we are interested in totally defined strategies, of course, but since we will be defining these objects using recursion we cannot avoid dealing with partial strategies.) For each world strategy ω and each partial agent strategy σ we define by recursion the (partial) sequences

$$s_0 = \omega(\varnothing),$$
$$(a_n, w_n) \simeq \sigma(\bar{s}(n)),$$
$$s_{n+1} \simeq \omega(\bar{a}(n)),$$

and the associated streams of acts and states

$$\omega * \sigma = a_0, a_1, ..., \tag{12}$$

$$\omega \bullet \sigma = s_0 a_0, s_1 a_1, ..., \tag{13}$$

terminated after $n+1$ acts by 1 if for some (first) n we have $w_n \simeq 1$. Two partial strategies are *equivalent* if they produce the same run against every ω; i.e.,

$$\sigma \sim \tau \Leftrightarrow (\forall \omega)[\omega * \sigma = \omega * \tau],$$

which implies that also $\omega \bullet \sigma = \omega \bullet \tau$. It is quite trivial to verify that very partial strategy σ is equivalent to exactly one *normalized* τ which satisfies the following conditions:

1. $\tau(\bar{s}(n))\downarrow \Rightarrow (\forall i < n)[\tau(\bar{s}(i))\downarrow]$.
2. $\tau(\bar{s}(n) \simeq (a_n, 1) \Rightarrow (\forall m > n, \bar{s}(m))[\tau(\bar{s}(m)) \simeq (skip, 1)]$.
3. If $\tau(\bar{s}(n)) \simeq (a_n, w_n)$ and the next state s_{n+1} is not accessible from $s_n a_n$, then $\tau(\bar{s}(n+1)) \simeq (skip, 1)$.

3.2. *Behaviors and behavior functions.* The set

$$\mathsf{B} = \mathsf{B}(\mathscr{S})$$

of *behaviors*[2] of a structure $\mathscr{S}$ consists of all the normalized partial, agent strategies in $\mathscr{S}$. It carries the natural partial ordering

$$\sigma \leqslant \tau \Leftrightarrow (\forall \bar{s}(n))[\sigma(\bar{s}(n))\downarrow \rightarrow \sigma(\bar{s}(n)) \simeq \tau(\bar{s}(n))]$$

with which it is a dcpo. A *behavior function* in $\mathscr{S}$ is any monotone and continuous

$$F\colon \mathsf{B}^n \rightarrow \mathsf{B}.$$

In a trivial structure with just the one state ι we will identify a behavior σ with the stream

$$\sigma(n) \simeq \begin{cases} a_n & \text{if } \sigma(i^n) \simeq (a_n, w_n) \text{ and } [n=0 \text{ or } \sigma(\iota^{n-1}) \simeq (a_{n-1}, \partial)], \\ 1 & \text{if } \sigma(\iota^n) \simeq (a_n, 1), \text{ otherwise.} \end{cases} \tag{14}$$

Behavior functions on trivial structures are thus identified with functions on streams.

[2] Park (1980) models a (total behavior by a sequence of multiple valued functions on the set of states where we use a sequence of functions $(\sigma_0, \sigma_1, \ldots)$, with σ_n: $\mathsf{States}^{n+1} \rightarrow \mathsf{Acts}$. Thus, apart from the detail that we do not identify acts with transition functions, the basic difference is that our agents are endowed with *local memory* of what they have seen in their previous interactions. The indeterminacy modeled by Park's use of multiple valued functions is handled here at the level of process modeling, but we could use *multiple valued strategies* at the outset without affecting any of our results.

Obvious behavior functions are *act execution*

$$a = do(a) = \lambda(\bar{s}(n)) \text{ if } n=0 \text{ then } (a, 1) \text{ else } (skip, 1),$$

and the *conditionals*

$$cond_R(\sigma, \tau)(\bar{s}(n)) = \begin{cases} \sigma(\bar{s}(n)), & \text{if } R(s_0), \\ \tau(\bar{s}(n)), & \text{if } \neg R(s_0), \end{cases}$$

one for each relation R on the states.

To define more complex behaviors and behavior functions we will typically describe informally how player II should respond to I's legal moves, without worrying about normalization. Consider, for example the *sequential execution* of two behaviors σ *next* τ. With the indicated assumptions about the moves of σ, the first few moves of II by σ *next* τ are as follows:

	I	s_0	s_1	s_2	s_3	s_4
σ:	II	(a_0, ∂)	(a_1, ∂)	$(a_2, 1)$		
σ *next* τ:	II	(a_0, ∂)	(a_1, ∂)	(a_2, ∂)	$\tau(s_3)$	$\tau(s_3, s_4)$

3.3. A *binary merger* on a state structure $\mathscr{S}$ is any (total) function μ: **States*** $\rightarrow \{0, 1\}$ on non-empty sequences of states. Given behaviors σ_0, σ_1 in $\mathscr{S}$, *the disjunctive merge* $\mu_\vee[\sigma_0, \sigma_1]$ *of* σ_0 *and* σ_1 *by* μ is defined (roughly) by decreeing that in a certain stage of the game it calls σ_0 or σ_1 accordingly as μ gives the value 0 or 1. Rather than give the formal definition which is somewhat technical, we indicate the first few moves of the play by $\mu_\vee[\sigma_0, \sigma_1]$ for given values of μ:

	I	s_0	s_1	s_2	s_3	s_4
μ:		1	0	1	1	0
$\mu_\vee[\sigma_0, \sigma_1]$:	II	$\sigma_1(s_0)$	$\sigma_0(s_1)$	$\sigma_1(s_0, s_2)$	$\sigma_1(s_0, s_2, s_3)$	$\sigma_0(s_1, s_4)$

We say that σ_i *is called at stage* n in a run of the game by $\mu_\vee[\sigma_0, \sigma_1]$ if the run is defined at least as far as stage n and $\mu(\bar{s}(n)) = i$. We say that σ_i *terminates at stage* n if it is called at stage n and yields a move of the form $(a, 1)$, in which case, of course, the run by $\mu_\vee[\sigma_0, \sigma_1]$ also terminates.

We define *the conjunctive merge* $\mu_\&[\sigma_0, \sigma_1]$ *of* σ_0 *and* σ_1 *by* μ in the same way, but add the stipulation that (as in the definition of *next*), if some σ_i terminates first, then the merged behavior changes 1 to ∂ and from that stage on calls the other behavior σ_{1-i} independently of the value of μ. For example:

	I	s_0	s_1	s_2	s_3	s_4
μ:		1	0	1	0	0
σ_0:	II		$(a_0, 1)$			
σ_1:	II	(b_0, ∂)		(b_1, ∂)	(b_2, ∂)	$(b_3, 1)$
$\mu_{\&}[\sigma_0, \sigma_1]$:	II	(b_0, ∂)	(a_0, ∂)	(b_1, ∂)	(b_2, ∂)	$(b_3, 1)$.

A merger μ is *state independent* if its values depend only on the stage of the run and not what has been played, i.e., for some $v: N \to \{0, 1\}$ and all sequences of states,

$$\mu(s_0, ..., s_n) = v(n),$$

and the simplest of these is the alternating sequence of 0's and 1's

$$\mu_0 = 0, 1, 0, ... \tag{15}$$

In practice, we will also need *n-ary mergers* μ: States* $\to \{0, ..., n-1\}$ and their disjunctive and conjunctive actions on n-tuples of behaviors $\mu[\sigma_0, ..., \sigma_{n-1}]$, defined in the same way. We will consider several kinds of mergers later, but for now we simply note that the functions defined by 3.3 and the corresponding n-ary functions are behavior functions, i.e., monotone and continuous.

3.4. A *behavior structure* is a pair $\mathscr{A} = (\mathscr{S}, \mathscr{F})$, where $\mathscr{S}$ is a state structure and $\mathscr{F}$ assigns an n-ary behavior function $\mathscr{F}(f)$ to each function symbol of arity n. We define the semantics of $\mathscr{L}$ on such structures by letting the variables vary over behaviors and interpreting the recursion construct again by least fixed point recursion, so that for each expression E and each n-tuple of variables $\vec{x}$ which includes all the free variables of E we obtain a behavior function

$$F_E = behavior(\mathscr{A}, \vec{x})E,$$

the *behavior denotation of* E *relative to* $\vec{x}$. If E is a *closed expression* (with no free variables), then its behavior denotation F_E relative to the empty list is a function with no arguments, i.e., just a behavior. Two expressions E, M are *behavior equivalent* if they have the same behavior denotation $F_E = F_M$ on every behavior structure and for every $\vec{x}$. We write:

$$E \equiv_{\text{beh}} M \Leftrightarrow E, M \text{ are behavior equivalent.}$$

3.5. *Interrupt handling.* To illustrate the difference between procedure

and behavior semantics, suppose that the state as an integer variable X and consider the closed expression *checkzero* defined by the recursion

$$checkzero = if\ X = 0\ then\ skip\ else\ \{skip\ next\ checkzero\}.$$

The procedure assigned to *checkzero* checks the state once, terminates if $X = 0$ and idles forever if $X \neq 0$; the behavior assigned to *checkzero* is the function on sequence of states such that

$$checkzero(s_0, ..., s_n) = if\ (\exists i \leqslant n)\ X(s_i) = 0\ then\ (skip, 1)\ else\ (skip, \partial).$$

In a "closed environment" where *checkzero* is the only program being executed both denotations will produce the same sequence of states. On the other hand, the behavior denotation of *checkzero* can be "played against" a world where other agents are changing the state and it will have the intended effect, i.e., idle until X becomes 0 and only then terminate. If the value stored in X records whether an interrupt has been received and *exit* is some behavior which arranges for ending execution in an orderly fashion, then for every behavior σ,

$$inthandle(\sigma) \equiv \mu_{0_\vee}\ [checkzero, \sigma]\ next\ [if\ (X = 0)\ then\ exit\ else\ skip]$$

modifies σ so that it checks for and handles interrupts. Here μ_0 is the simplest fair, alternating merger of (15).

3.6. *Synchronization.* One should not understand the polite exchange of moves by an agent X and "the rest of the world" as providing somehow a model of an asynchronous, concurrent system. Our image of such a system is one of asynchronous chaos: many agents are operating on the same state at randomly chosen, unrelated times, some acts being executed "simultaneously," others requiring some non-zero time interval for their completion. Someone—an operating system or "nature"—sees to it that truly incompatible acts are not executed simultaneously, but short of that, anything goes. The polite exchange of moves s_0, (a_0, w_0), ... is an idealized version of the agent's X perception of what is happening, and it is (by its nature) a one-dimensional *sequence of frames* (still pictures) of the actual, fluid situation. The agent X can act, we must assume this; and when he executes an act a_n he is reacting to some state s_n, or whatever part of it he can see. The actual state of the world is probably changing even as a_n is being executed, except of course this change cannot affect the execution of a_n (or that execution could not have happened), it will affect X's next act.

These idealized frame sequences are this modeling's solution of the *synchronization problem* for dependence and causal connection among acts,

which in other concurrency modelings is solved by partial orders (Pratt, 1986), matching pairs of inverse acts (Hoare, 1978; Milner, 1979, 1983), etc. There is no common clock, but if we are both trying to reverse the last seat on the night plane to Washington, it matters who gets to the Airline first: this "mattering" is coded (in different ways) in our respective, frame-sequence views of the changing world. The modeling does not attempt to provide a global picture of the entire, concurrent system; all we have are the frame sequences observed by the individual agents.

Fix now a procedure structure $\mathscr{A}$. With each procedure

$$\alpha = \lambda(s)(\alpha(s)(0), \alpha(s)(1), \dots)$$

in $\mathscr{A}$ we associate the behavior $\iota(\alpha)$ defined by

$$\iota(\alpha)(\bar{s}(n)) \simeq \begin{cases} (\alpha(s_0)(n), 1) & \text{if } [\alpha(s_0)(n+1) \simeq 1], \\ (\alpha(s_0)(n), \partial) & \text{otherwise}, \end{cases}$$

which checks out only the first state s_0 it sees and then blindly plays the moves dictated by $\alpha(s_0)$. Conversely, with each behavior σ we associate the procedure $\iota^{-1}(\sigma)$ defined by

$$\iota^{-1}(\sigma)(s) = \omega_s * \sigma,$$

where $\omega_s(\emptyset) = s$, $\omega_s(\bar{a}(n)) = s\bar{a}(n)$ and the stream function $*$ is defined by (12). Obviously

$$\iota^{-1}(\iota(\alpha)) = \alpha,$$

but the composition in the opposite order is not the indentity because the map ι^{-1} "loses information." Using these maps we can lift faithfully procedure functions to behavior functions, setting (in the case of a procedure function of one variable)

$$\phi^{\iota}(\sigma) = \iota(\phi(\iota^{-1}(\sigma))).$$

It is trivial to verify that this interpretation of procedure semantics in behavior semantics is faithful, and the (converse) second part of the next theorem is also quite easy.

THEOREM 3.7. *Let $\mathscr{A}^{\iota}$ be the behavior structure associated with a procedure structure $\mathscr{A}$ by replacing each procedure function ϕ in $\mathscr{A}$ by the corresponding behavior function ϕ^{ι}. For every expression E then, if ϕ_E and F_E are the functions asociated with E in $\mathscr{A}$ and $\mathscr{A}^{\iota}$, respectively,*

$$\phi_E(\alpha_1, \dots, \alpha_n) = \iota^{-1} F_E(\iota(\alpha_1), \dots, \iota(\alpha_n))$$

for all procedures $\alpha_1, ..., \alpha_n$. *As a consequence, behavior equivalent expressions are also procedure equivalent.*

Conversely, if two expressions are procedure equivalent, then they are also behavior equivalent.

This result is announced in Moschovakis (1990), where in fact it is proved for a much stronger language, with fixed, standard interpretations of act execution, sequential execution, and conditionals. It may be interpreted as saying that *the logic of* $\mathscr{L}$ (and the stronger language of Moschovakis (1990) *does not change when we switch from a non-interactive to an interactive interpretation.*

4. Players, Types, and Merges

4.1. *Players.* The behaviors of a state structure $\mathscr{S}$ model deterministic, interactive programs and correspond to the Park (1980) *abstract paths* and the ***behaviors*** of Pratt (1986). To model non-deterministic processes we will use the set of *players*

$$\mathscr{P} = \mathscr{P}(\mathscr{S}) = \{x \subseteq \mathsf{B}(\mathscr{S}) \mid x \neq \varnothing\},$$

the analog in this approach of the *processes* of Pratt (1986) or the (parameter-free) *agents* of Milner (1979, 1983). Thus we follow one of the general methods in this area of modeling a process by the set of all the behaviors it may exhibit, the set of partial strategies representing a process in our model capable of "playing" the game of interaction using any of its members. Notice, however, that the non-determinism we allow is only *initial*: a player may choose any of his behaviors at the beginning of the game, but then he must stick with it for the whole run. A player is *deterministic* if he is a singleton, *total* if all his behaviors are total, and *convergent* if every run of G by one of his members terminates at a finite stage. In a trivial structure a player is just a *non-deterministic stream*, i.e., a non-empty set of streams.

If x is a player and Ω is a set of world strategies, we let

$$\Omega * x = \{\omega * \sigma \mid \omega \in \Omega, \sigma \in x\}$$

be the set of all streams of acts which can result by pitting x against the external world represented by Ω. When $\Omega = \{\tilde{\omega}\}$ represents an inert external world, we get the nondeterministic stream

$$path(x) = \{\tilde{\omega}\} * x = \{\tilde{\omega} * \sigma \mid \sigma \in x\} \tag{16}$$

of all "isolated behaviors" of x.

The *sequential composition* of players is defined by

$$x \; next \; y = \{\sigma \; next \; \tau \mid \sigma \in x, \tau \in y\},$$

which illustrates a general method of extending behavior functions to player arguments by "distributing":

$$F(x_1, ..., x_n) = \{F(\sigma_1, ..., \sigma_n) \mid \sigma_1 \in x_1, ..., \sigma_n \in x_n\}.$$

We can extend to players in this way act execution, conditionals, etc. Non-deterministic players are introduced directly with the disjunction operation

$$x \; or \; y = x \cup y.$$

4.2. More interesting are the *merge operations*

$$par_M(x_1, ..., x_n) = \langle \mu_{\&}[\sigma_1, ..., \sigma_n] \mid \sigma_1 \in x_1, ..., \sigma_n \in x_n, \mu \in M\}, \quad (17)$$

one for each set $M \neq \emptyset$ of n-ary mergers. For example, the full (unfair, binary) merge is defined by

$$merge(x, y) = \{\mu_{\&}[\sigma, \tau] \mid \sigma \in x, \tau \in y, \mu \text{ any merger}\}.$$

In the interesting applications we choose M by restricting μ to satisfy various *safety* and *liveliness* conditions, e.g. freedom from deadlock, fairness, etc. To deal with such properties we introduce the basic notion of (act) type of a player.

4.3. A behavior τ is *consistent with* a set of behaviors a if for every finite sequence of state sequences $\bar{s}_0(n_0), ..., \bar{s}_k(n_k)$,

$$(\forall i \leqslant k)[\tau(\bar{s}_i(n_i))\downarrow] \Rightarrow (\exists \sigma \in \mathsf{a})(\forall i \leqslant k)[\tau(\bar{s}_i(n_i)) \simeq \sigma(\bar{s}_i(n_i))].$$

If we think of the set a as coding a property of behaviors, then the τ's consistent with a are those which we cannot recognize as not having the property a by any finite set of "independent, terminating experiments."

A *type*[3] (of behavior) is a set of behaviors a which contains all the behaviors consistent with it. For each type a, we let

$$\mathscr{P}(\mathsf{a}) = \{x \mid \emptyset \neq x \subseteq \mathsf{a}\}$$

be the set of all players *of type* a. The totally undefined behavior $\emptyset$ is

[3] Types are the non-empty closed sets in the natural (non-Hausdorff) topology on B viewed as a set of partial functions. In Moschovakis (1989b) we used a more refined notion of type which comes from the stronger Baire topology, where we view behaviors as total functions into $(\mathsf{Acts} \times \{\partial, 1\}) \cup \{\perp\}$. The present definition identifies types with safety properties and covers all the interesting examples.

consistent with every set of behaviors, so it belongs to every type. The (smallest) *type of a player* x is defined by

$$\mathsf{type}(x) = \{\tau \mid \tau \text{ is consistent with } x\},$$

and for every type a, obviously,

$$x \in \mathscr{P}(\mathsf{a}) \Leftrightarrow \mathsf{type}(x) \in \mathscr{P}(\mathsf{a}).$$

If $x = \{\sigma\}$ is a deterministic player, then $type(x)$ is the set of all "sub-behaviors" of σ.

4.4. For each set of acts E,

$$\mathsf{eff}(E) = \{\sigma \in \mathsf{B} \mid (\forall \bar{s}(n), a, w)[\sigma(\bar{s}(n) \simeq (a, w) \Rightarrow a \in E\}.$$

A player of *effect type* $\mathsf{eff}(E)$ never executes an act outside of E. Note that $\mathsf{eff}(\varnothing) = \{\varnothing\}$.

4.5. For each set of acts D and each act a, put

$$res_D(a) = \begin{cases} a, & \text{if } a \in D, \\ skip, & \text{otherwise,} \end{cases} \tag{18}$$

and extend this *restriction map* canonically to histories by

$$res_D(a_1 \cdots a_n) = res_D(a_1) \cdots res_D(a_n).$$

We set

$$\mathsf{dep}(D) = \{\sigma \in \mathsf{B} \mid (\forall h_0, ..., h_n)[\sigma(\imath h_0, ..., \imath h_n) \simeq \sigma(\imath\, res_D\, h_0, ..., \imath\, res_D\, h_n)\},$$

where $h_0, ..., h_n$ are arbitrary histories so that $\imath h_0, ..., \imath h_n$ are arbitrary states accessible from $\imath$. The acts executed by a player of *dependence type* $\mathsf{dep}(D)$ depend only on "the part of the state" which can be changed by the execution of acts in D.

4.6. A binary merger μ is *fair* on a world strategy ω and behaviors σ, τ if either the stream of acts $\omega * \mu_{\&}[\sigma, \tau]$ in the game of ω versus the marged behavior is finite, or it is infinite and each of σ, τ either terminates or is called infinitely often; it is fair on a set Ω of world strategies and types a, b if it is fair on all $\omega \in \Omega$, $\sigma \in \mathsf{a}$, $\tau \in \mathsf{b}$. (The terminology is set in 3.3 and the definitions extend directly to *n-ary* mergers.)

The simplest useful examples of fair binary merges are

$$fairmerge(x, y) = \{\mu_{\&}[\sigma, \tau] \mid \sigma \in x, \tau \in y, \mu \text{ fair on all } \omega, \sigma', \tau'\},$$

$$parkmerge(x, y) = \{\mu_{\&}[\sigma, \tau] \mid \sigma \in x, \tau \in y, \mu \text{ state independent and fair on all } \omega, \sigma', \tau'\}.$$

We have used the term "parkmerge" for the *fair merge* of Park (1983), his terminology being more appropriate in our context for the full, state dependent fair merge. Assuming that the "while" construct will be interpreted correctly by our treatment of recursion, it is clear that both this *parkmerge* and *fairmerge* will give reasonable interpretations of Park's example 1.1, the full *fairmerge* allowing a richer set of "cleverer" scheduling algorithms which takes the sequence of states ω played by the world into account in deciding how long to wait before calling for the assignment $X := 0$ to be executed.

4.7. *Dijkstra's example, revisited.* In the structure appropriate for 1.2, a state assigns each philosopher arbitrarily to eat or think, and we have acts eat_i and $think_i$ which change the state in the obvious way; initially all philosophers are thinking. A state is *possible* if it does not assign any two adjacent philosophers to eat, otherwise it is *deadlocked.* From the definition of player recursion which we w:ill give in the next section, it will be clear that (as expected) the player interpretation of each expression D_i in (1) is a deterministic player with sole member a behavior δ_i which moves alternatively $think_i$, eat_i, independently of the state. Let

$$\mathsf{d}_i = type(\{\delta_i\}),$$

and let d be the type of behaviors which produce no deadlock against the inert world strategy,

$$\delta \in \mathsf{d} \Leftrightarrow \tilde{\omega} \cdot \delta \text{ is a stream of possible states.}$$

(The definitions of $\tilde{\omega}$ and $\cdot$ are in (11) and (13).) We define the set M of mergers on five arguments by

$$\begin{aligned} \mu \in M \Leftrightarrow \mu &\text{ is fair on } \mathsf{d}_0, ..., \mathsf{d}_4 \text{ against } \tilde{\omega} \\ &\text{and } (\forall \delta_0 \in \mathsf{d}_0, ... \, \delta_4 \in \mathsf{d}_4)[\mu_{\&}(\delta_0, ..., \delta_4) \in \mathsf{d}]. \end{aligned}$$

It is clear that the fair, non-deadlock interpretation of Dijkstra's example is the non-deterministic stream of acts

$$D = \tilde{\omega} * par_M(D_0, ..., D_4) \tag{19}$$

defined from this M, according to (17).

Clearly (19) gives only a specification of the set of acceptable scheduling algorithms and contributes nothing to the problem of constructing clever, efficient or in any way special schedulers for Dijkstra's problem. We do not consider this a defect of our approach, as we take it for granted here that this is all that correct semantics can do: they can make "algorithmic" problems precise, but we cannot expect that they will solve them.

5. Non-deterministic Networks[4]

One version of the so-called *merge anomaly* of example 1.3 shows up when we try to visualize the function

$$f(M) = fairmerge(5, S(M))$$

and its fixed point $\bar{M}$ as *non-deterministic stream networks*, in the most obvious, naive manner. Figure 1(a) specifies the network

$$O = fairmerge(5, S(M))$$

with input stream M and output stream O, and Fig. 1(b) specifies the network

$$M = fairmerge(5, S(M))$$

with no input.

There is nothing problematical about the first of these diagrams, which appears to give an accurate, network picture of the non-deterministic function f: for each (possibly non-deterministic) stream M, $O = f(M)$ will produce all fair merges of the one-act stream $(5, 1)$ and streams in $S(M)$. On the other hand, there are two natural "naive" ways to understand the network diagram 1(b).

5.1. *First understanding.* If we understand naively the merge operation in 1(b) to be "strict", then the output is

$$\bar{M}_1 = \{\varnothing, (5, 6, ...)\},$$

where the empty stream $\varnothing$ is output if the "implementing" merger μ looks for input first on the right and $(5, 6, ...)$ is output if μ looks for input first on the left—and then never looks left again, since 5 is a terminating stream with just one element.

5.2. *Second understanding.* If we take the merge in 1(b) to be "non-strict"—or *angelic* in Broy's terminology—so that the merge node will necessarily produce output if either of its inputs brings in something, then 5 (and the information that this input is complete) is eventually received on the left, and then the output will be the (deterministic) stream

$$\bar{M}_2 = \{(5, 6, ...)\}.$$

The "anomaly" is that neither of these interpretations produces a fixed

[4] This section is not used in the sequel and may be skipped by those more interested in the theory of recursion which follows.

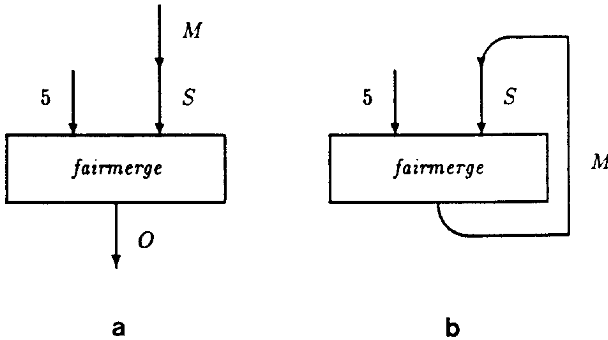

FIGURE 1

point of the equation (3) $M = f(M)$. To understand the cause of the problem, let us consider the similar-looking equation

$$K = (0 \textit{ or } 1) \textit{ next } K,$$

where the non-determinism is put in the (constant) stream 0 *or* 1. Now the merge is not involved and surely the most natural fixed point of this equation is

$$\bar{K} = \text{all infinite sequences of 0's and 1's.}$$

We can draw again, however, the pictures for the networks

$$O = (0 \textit{ or } 1) \textit{ next } K, \qquad K = (0 \textit{ or } 1) \textit{ next } K$$

(see Fig. 2), and by staring at those and using the same intuitions about networks which led to $\bar{M}_1$ and $\bar{M}_2$ above we should arrive at the much smaller value for the output of Fig. 2(b)

$$\bar{K}_0 = \{\bar{0}, \bar{1}\} = \{(0, 0, ...,), (1, 1, ...)\}.$$

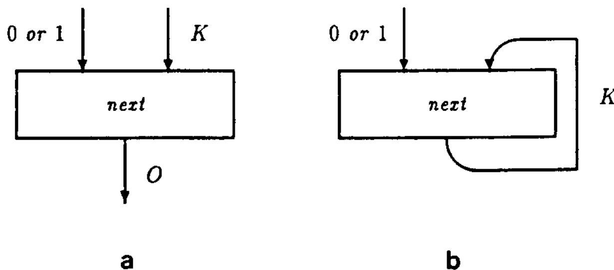

FIGURE 2

The analogy between these two examples suggests that "the merge anomaly" (at least this version of it) is not really about the merge operation, but about our understanding of non-deterministic streams in general. Broy (1986) gives a similar analysis and concludes that *recursive, non-deterministic stream definitions must be interpreted as systems of recursive, deterministic stream definitions.* Our conclusion is (in part) that definitions of networks like the above do not involve recursion at all but only a form of "indirect self-reference", and that the modeling must take into more explicit account the implicit assumption that communication between streams is by buffers.[5]

5.3. For each non-empty set of acts E, the behaviors of type

$$\mathsf{stream}(E) = \mathsf{dep}(\emptyset) \cap \mathsf{eff}(E) \tag{20}$$

are "blind" to changes in the state and cannot execute an act outside E, so they can be identified with streams of actions in E. This is done formally ia the one-to-one function

$$stream(\sigma) = \lambda(n) \begin{cases} a_n & \text{if } \sigma(\imath^n) \simeq (a_n, w_n) \;\&\; [n = 0 \text{ or } \sigma(\imath^{n-1}) \simeq (a_{n-1}, \partial)], \\ 1 & \text{if } \sigma(\imath^n) \simeq (a_n, 1), \text{ otherwise}, \end{cases}$$

but in practice we will use σ and $stream(\sigma)$ synonymously when $\sigma \in \mathsf{stream} = \mathsf{stream}(\mathsf{Acts})$. Ditto for the non-deterministic E-streams, i.e., players of type $\mathsf{stream}(E)$, to which we extend the map above by distributing

$$stream(x) = \{stream(\sigma) \mid \sigma \in x\}.$$

In modeling networks we will use extensively the types of extended streams

$$\mathsf{extream}(E) = \mathsf{stream}(E \cup \{skip\}), \tag{21}$$

which in addition to executing acts in E may "pass" (or "stutter"), i.e., output *skip*.

5.4. *Restriction.* The map res_E of (18) can be extended to behaviors by

$$res_E(\sigma) = \lambda(\bar{s}(n)) \begin{cases} (res_E(a), w), & \text{if } \sigma(\bar{s}(n) \simeq (a, w), \\ \text{undefined}, & \text{if } \sigma(\bar{s}(n) \text{ is undefined}, \end{cases}$$

[5] There is nothing new to these ideas which can all be found in one form or another in Park (1983). The remainder of this section may be viewed as a development of Park's analysis, perhaps providing some natural justification for the introduction of *hiatons* which seems somewhat *ad hoc* in Park (1983).

so that (immediately) $res_E(\sigma) \in \mathsf{eff}(E \cup \{skip\})$. We will use the same notation res_E for the function on players

$$res_E(x) = \{res_E(\sigma) \mid \sigma \in x\}.$$

5.5. *The essence* $ess(\alpha)$ of a stream of acts α (according to Park) is the stream obtained by deleting all occurrences of *skip* in α, except that $ess(skip, 1) = (skip, 1)$ since we have not allowed the empty, convergent stream. We extend this function to behaviors by

$$ess(\sigma) = stream^{-1}\, ess\, stream(\sigma),$$

and further to players by distributing. Note that for every x, $ess(x) = \{ess(\sigma) \mid \sigma \in x\}$ is of type $\mathsf{eff}(\mathsf{Acts}\setminus\{skip\})$,

5.6. Adapting Kahn (1974) to our framework, a *program schema* is a finite oriented graph G where edges need not have nodes at both ends and where each edge e is labeled with *a set of acts* $D(e)$, intuitively the acts of sending messages of a ertain kind along e. It is assumed that

$$e \neq e' \Rightarrow [D(e) \cap D(e') = \varnothing].$$

Inputs are the edges with no beginning node, *outputs* are those with no ending node, and the *inputs* of each edge e are the edges ending at the beginning node of e. In addition to the input edges, there may also be non-input edges (with a beginning node) which have no inputs. We will identify a program schema with the state structure whose acts are those in the $D(e)$'s and *skip* (assumed not in any $D(e)$), and where each state is a function which assigns to each edge e a *history* $H(e) \in (D(e) \cup \{skip\})^*$ of acts in $D(e) \cup \{skip\}$, representing what has already occurred along e. In the initial state all these histories are empty.

5.7. For each edge e of a program schema G, let

$$\mathsf{indep}(e) = \mathsf{dep}\left(\bigcup \{D(e') \mid e' \text{ is an input edge of } e\}\right)$$

be the dependence type of possible acts on the input edges to e. A *network* on a program schema G is an assignment

$$\mathcal{N} = \{N(e) \mid e \text{ an edge of } G\}$$

to each edge of a *total player* $N(e)$ of type

$$T(e) = \mathsf{indep}(e) \cap \mathsf{eff}(D(e) \cup \{skip\}).$$

The network is *deterministic* if each $N(e)$ is deterministic, i.e., the singleton

$N(e) = \{\sigma_e\}$ of a behavior, otherwise it is *non-deterministic.* Note that by this definition, if e has no input edges, then $N(e)$ is a (possibly) non-deterministic, extended $D(e)$-stream.

If the edge e has input edges $e_1, ..., e_n$ then the player $N(e)$ assigned to e induces a function

$$f_e : \mathscr{P}(\text{extstream}(D(e_1))) \times \cdots \times \mathscr{P}(\text{extstream}(D(e_n))) \\ \rightarrow \mathscr{P}(\text{extstream}(D(e)))$$

by the formula

$$f_e(x_1, ..., x_n) = \textit{path fairmerge}(N(e), x_1, ..., x_n), \tag{22}$$

where *path* is defined by (16). These extended stream functions are adaptations to our setup of Park's $>$-functions in (Park, 1983); they make explicit the assumption that *communication along channels is by buffers* and their use is characteristic of the present modeling. There is an implicit appeal to them in the next key definition.

5.8. The *life* of a network $\mathcal{N}$ with players $N(e_1), ..., N(e_m)$ assigned to its edges is the extended stream

$$\textit{life}(\mathcal{N}) = \textit{path fairmerge}\ (N(e_1), ..., N(e_m)). \tag{23}$$

The *stream system* defined by the network is the set of tuples

$$\overline{\text{life}}(\mathcal{N}) = \{(\textit{ess res}_{D(e_1)}(\sigma), ..., \textit{ess res}_{D(e_m)}(\sigma)) \mid \sigma \in \textit{life}(\mathcal{N})\}, \tag{24}$$

and *the* (non-deterministic *stream determined by the network along each edge e* is

$$\textit{str}(N)_e = \textit{ess res}_{D(e)}(\textit{life}(N)).$$

Directly from the definitions, we have

$$\text{if } e \text{ has no inputs, then } \textit{str}(N)_e = \textit{ess}(N(e)).$$

It is natural to think of the inputs to a network as *variables* and its other players as *constants*, so that the life and the streams of the network are functions of its inputs.

The use of *fairmerge* in (23) is of course the key element of this modeling, which represents a network by all fair interleavings of the activity in the nodes. The discussion in 3.6 should make it clear that our analysis does not exclude "simultaneous" sending of messages between nodes, it only codes it by the many possible private views of such activity at the individual nodes. The use of *player types* here gives a concrete

example of how changes in the state which may be happening even while an agent (node) is acting must not interfere with his acts, but may affect his subsequent actions, as discussed rather vaguely in 3.6. Note also that the modeling of each node by a total player gives the most direct non-deterministic semantics to the simple programming language Kahn (1974) uses to motivate his results.

Before we go into the representation of Example 1.3 in this modeling, let us consider (following Park, 1983) a kind of "converse" to definition (22), i.e., how we can represent an arbitrary monotone, continuous function on streams by a total player. Suppose

$$F\colon \mathsf{stream}(D(e_1)) \times \mathsf{stream}(D(e_2)) \to \mathsf{stream}(D(e))$$

is monotone and continuous with two arguments (for simplicity), and define $\tau_F(\bar{s}(n))$ by recursion on the length $n+1$ of the sequence of states $\bar{s}(n)$,

$$\tau_F(\bar{s}(n)) = \begin{cases} \text{the first act in } F(H(s_n, e_1), H(s_n, e_2)) \text{ not in } \bar{t}^n, & \text{if such exists,} \\ \mathit{skip}, \quad \text{otherwise,} \end{cases} \tag{25}$$

where $H(s_n, e_i)$ is the history of acts along e_i coded in the state s_n and viewed as a finite stream (possibly with the terminator 1 at the end) so we can plug it into F, and

$$\bar{t}^n = \mathit{ess}(\tau_F(\bar{s}(0)), \tau_F(\bar{s}(1)), \ldots, \tau_F(\bar{s}(n-1))).$$

Clearly τ_F is a total strategy and *if* e_1, e_2, e *are all distinct edges* (so that the acts along them are distinct), then for all $\sigma_1 \in \mathsf{stream}(D(e_1))$, $\sigma_2 \in \mathsf{stream}(D(e_2))$,

$$F(\sigma_1, \sigma_2) = \mathit{ess}\ \mathit{res}_{D(e)} \mathit{fairmerge}(\{\tau_F\}, \{\sigma_1\}, \{\sigma_2\}\}). \tag{26}$$

On the other hand, if $e = e_1$, then the right-hand-side of (26) involves an "indirect self-reference" which amounts to recursion and we can verify that for all σ_2,

$$\mathit{ess}\ \mathit{res}_{D(e)} \mathit{fairmerge}(\{\tau_F\}, \{\sigma_2\}\}) = \{\text{the least fixed point of } F(\sigma_1, \sigma_2) = \sigma_1\}.$$

We can use this construction and straightforward least-fixed-point arguments to show that the present modeling of networks extends conservatively the semantics of Kahn, Park, and Broy.

THEOREM 5.9. *Suppose G is a program schema, for each non-input edge e we are given a set $\mathscr{F}_e$ of monotone, continuous stream functions of type*

$$F\colon \mathsf{stream}(D(e_1)) \times \cdots \times \mathsf{stream}(D(e_n)) \to \mathsf{stream}(D(e))$$

on the input edges of e, and for each input edge e of G we are given a non-deterministic stream $x_e \in \mathscr{P}(\mathsf{stream}(D(e)))$. Define a network by assigning the x_e's to the input edges and to each non-input edge e the total player

$$N(e) = \{\tau_F \mid F \in \mathscr{F}_e\},$$

where each τ_F is defined as in (25), and let life($\mathscr{N}$) be the extended stream describing this network. A tuple $(\sigma_1, ..., \sigma_m)$ of streams is in the stream system of this network defined by (24) if and only if $(\sigma_1, ..., \sigma_m)$ is the sequence of simultaneous least fixed points of some system

$$\begin{gathered} \sigma_1 = F_1(\sigma_1, ..., \sigma_m). \\ \vdots \\ \sigma_m = F_m(\sigma_1, ..., \sigma_m), \end{gathered}$$

where $F_1, ..., F_m$ is an arbitrary choice of functions, $F_i \in \mathscr{F}_{e_i}$, $i = 1, ..., m$.

In particular, if each $\mathscr{F}_e$ is a singleton and the input edges are assigned deterministic streams, then the hypothesis describes an arbitrary Kahn network and the conclusion asserts that its representation in this modeling yields the stream system assigned to it by the Kahn semantics. The general case shows similar agreement with Broy's modeling, for networks without his ambiguity operator.

THEOREM 5.10. *Suppose G is a program schema with non-input edges "of two kinds," for each input edge and each edge e of the first kind we are given a stream or a set of functions $\mathscr{F}_e$ as in 5.9, and for each edge e of the second kind we are given a merge operation on two specified input edges e^1, e^2 of e,*

$$f_e(x_1, x_2) = \mathit{fairmerge}(x_1, x_2).$$

Define a network by assigning to each edge e of the first kind a total player as in 5.9 and to each edge e of the second kind the total player

$$N(e) = \mathit{fairmerge}(\{\tau_1\}, \{\tau_2\}),$$

where τ_i is assigned to the identity function I_i along e^i as in (25). It follows that for every tuple $(\sigma_1, ..., \sigma_m)$ of streams in the stream system of this network, there is a choice of functions $F_i \in \mathscr{F}_i$ for each edge e_i of the first kind so that

$$e_i \text{ of the first kind} \Rightarrow \sigma_i = F_i(\sigma_1, ..., \sigma_m);$$

and if e_i is of the second kind, then σ_i is a full (non-strict, angelic) merge of the appropriate inputs σ^1, σ^2, i.e., it can be decomposed into two disjoint copies of σ^1 and σ^2.

Proof. We choose (strict, fair) mergers at the edges of the second kind, use a direct fixed point argument as in the preceding theorem and then chase the result of applying the essence function. ▌

This last theorem is the easy part of the main result of Park (1983), which explains how Park's modeling of "non-strict, fair (angelic) merge" in terms of strict, fair merge and hiatons can be adapted to our setup. It applies to the "anomalous" equation

$$M = fairmerge(5, S(M))$$

which may be viewed as defining a Park network which has two edges of the first kind with the single stream functions 5 and S attached to them, and just one merge edge. The stream determined by this network on the merge edge is

$$\bar{M}_2 = \{(5, 6, ...)\},$$

the stream which comes from the second naive understanding of the network in 5.2.

One can also formulate in this setting and prove the full strong result in Park (1983), which gives a "computational justification" of the method, but this is quite lengthy and we will skip it, especially as it does not add much to what is already in Park's paper.

6. Minimum Conditions for Process Smantics

Having settled on players to model non-deterministic processes, we now seek the appropriate objects to model *process transformations*. Arbitrary process functions $f: \mathscr{P}^n \to \mathscr{P}$ will not do because (for one thing) we will want fix-point equations of the form $x = f(x)$ to have solutions. The usual way of insuring the existence of fixed points by choosing the *monotone*, *continuous* player functions also does not work, since there is no obvious way to turn $\mathscr{P}$ into a dcpo. In fact we have been unable to define a reasonable process semantics for $\mathscr{L}$ which models process transformations with any set of *player functions* (which includes the fair merge operation), and we will use *intensional operations* which are not determined by their graphs. Now we will attempt to motivate and justify our choice of this modeling in the next section, but it is admittedly an unusual option. To make clear just what is achieved by the construction in the next two

sections, we set down here minimal conditions of "reasonableness" which must be satisfied by any process semantics for $\mathscr{L}$ with the fair merge.

6.1. Fix a state structure $\mathscr{S}$ and a signature τ which includes function symbols $do(a)$ for each act a of $\mathscr{S}$, *next*, $cond_R$ (R some relation on the states), *or*, and *fairmerge*. An abstract interpretation of $\mathscr{L} = \mathscr{L}(\tau)$ over $\mathscr{S}$ is a triple

$$\mathscr{A} = (\{\Phi_n\}_{n \in N}, \{\chi_n\}_{n \in N}, den)$$

which satisfies the following conditions.

1. Each *extension map* $\chi_n : \Phi_n \to (\mathscr{P}^n \to \mathscr{P})$ associates an n-ary player function $\chi_n \phi$ with each *abstract* (intensional) *function* $\phi \in \Phi_n$. It is of course allowed that each Φ_n is a set of n-ary player functions and χ_n is the identity.

2. The *denotation* map *den* assigns to each expression E of $\mathscr{L}$ and each list $\vec{x} = x_1, ..., x_n$ of distinct variables which includes all the free variables of E, an abstract n-ary function $den(\vec{x})E \in \Phi_n$.

3. If M is obtained from E by an alphabetic change of the bound variables and the (free) replacement of each x_i by some z_i, then $den(\vec{x})E = den(\vec{z})M$. In particular, each n-ary function symbol of τ is assigned a unique abstract n-ary function $\bar{f} = den(\vec{x})\, f(\vec{x})$ (with any choice of $\vec{x}$) and a player function $\bar{f} = \chi_n \bar{f}$.

This basic definition allows trivial interpretations which (for example) may assign the same object to all closed expressions. We call an abstract interpretation *minimally reasonable* if in addition it satisfies the following three conditions.

4. *Correctness for substitution.* If an m-ary function letter f is assigned the player function $\bar{f}$ and $\chi_n den(\vec{x})E_i = g_i$ for $i = 1, ..., m$, then

$$\chi_n den(\vec{x})\, f(E_1, ..., E_m) = \lambda(\vec{x})\, \bar{f}(g_1(\vec{x}), ..., g_m(\vec{x})).$$

5. *Soundness for recursion.* If two expressions E, M are least-fixed-point equivalent, then for all $\vec{x}$, $den(\vec{x})E = den(\vec{x})M$.

6. *Adequacy.* If f is one of the specified function symbols $do(a)$, *next*, $cond_R$, *or*, and *fairmerge* in the signature, then $\bar{f}$ is the associated player function, as we defined these in the last section.

The minimally reasonable interpretations of $\mathscr{L}$ cannot be entirely trivial, since they interpret correctly the explicit part of the language and they interpret mutual recursion (at least) by the taking of fixed points. On the other hand, *which fixed points* is not specified by the conditions above, and there may well exist minimally resonable interpretations which are

thoroughly unreasonable as concurrency modelings. The interpretation we will construct will be much more than minimally reasonable. On the other hand, I am not aware of any other concurrency models which combine the fair merge with full recursion and which are minimally reasonable in this sense, even in the special case where the underlying state structure is trivial (with just the one state ι), so that procedures and behaviors are just streams of acts and players are sets of streams (traces).

7. Modeling Process Transformations

The basic plan is to model a process transformation by the set of its "implementations," but it must be refined in two ways.

1. We will use a natural but unfamiliar notion of "abstract implementation" which may depend on an infinite sequence of arguments.

2. Instead of considering *extensionally* functions which are determined by *all* their implementations, we will adopt an *intensional approach* which identifies a function with *a specific* (suitably closed) *set* of implementations.

There is an obvious notion of an abstract implementation for functions on players which (in similar contexts) has been suggested by many, including Park and Broy: if, for example

$$f: \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$

is a binary player function,[6] then an abstract implementation for f would be any *monotone, continuous function*

$$F: \mathsf{B} \times \mathsf{B} \to \mathsf{B}$$

such that $\sigma \in x, \tau \in y \Rightarrow F(\sigma, \tau) \in f(x, y)$. For each of the functions *next*, *or*, par_M, and many others, there is a set I of such abstract implementations which determines the function, i.e.,

$$f(x, y) = \{F(\sigma, \tau) \mid \sigma \in x, \tau \in y, F \in I\}. \tag{27}$$

This suggests that we admit in our modeling only functions which satisfy (27) with some I. Consider however the function

$$twice(x) = x \; next \; x. \tag{28}$$

[6] We will often use unary and binary functions to illustrate definitions and results about arbitrary n-ary functions.

To compute *twice*(x) we (may) need to "call x twice", and there is no guarantee that the same behavior from x will be produced both times, if indeed x is a non-deterministic player. Thus the most natural implementation of this function is the operation of two arguments

$$H(\sigma, \tau) = \sigma \; next \; \tau, \tag{29}$$

and we have $twice(x) = \{H(\sigma, \tau) \mid \sigma, \tau \in x\}$. Going one step further, consider the recursion

$$y = x \; next \; y$$

which defines $\bar{y}(x)$ for each x. Clearly $\bar{y}(x) = x \; next \; x \; next \; x \cdots$, the computation of $\bar{y}(x)$ calls x infinitely many times if x is a conveergent player, and there is no reason why the same behavior from x will be called every time; if x is an infinite set of single act executions, then our intuition tells us that every infinite sequence of executions of acts from x is a possible value of $\bar{y}(x)$. Thus, to accommodate identification of variables and recursive definitions we must admit implementations of functions on players which depend on infinitely many arguments.

7.1. An *infinitary behavior function* (or just *behavior function* when no confusion can arise) is any monotone and continuous (in the obvious sense)

$$F\colon (N \to \mathsf{B})^n \to \mathsf{B},$$

which assigns to each n-tuple of players $\vec{x} = x_1, ..., x_n$, the *player value*

$$\begin{aligned} F(\vec{x}) &= \{F(p_1, ..., p_n) \mid p_1, ..., p_n\colon N \to \mathsf{B}, range(p_1) \subseteq x_1, ..., range(p_n) \subseteq x_n\} \\ &= \{F(p_1, ..., p_n) \mid p_1\colon N \to x_1, ..., p_n\colon N \to x_n\}. \end{aligned}$$

The first of these two equivalent expressions for $F(\vec{x})$ explains the idea better, but the second is more convenient and we will generally prefer it. We set

$$\mathsf{BF}_n = \{F \mid F\colon (N \to \mathsf{B})^n \to \mathsf{B}\},$$

$$\mathsf{B} = \bigcup_n \mathsf{BF}_n,$$

and we call $F \in \mathsf{BF}_n$ an *abstract implementation* of a player function $f\colon \mathscr{P}^n \to \mathscr{P}$, if $(\forall \vec{x})[F(\vec{x}) \subseteq f(\vec{x})]$. A player function f is *determined* by a set $I \subseteq \mathsf{BF}_n$ if for all $\vec{x} = x_1, ..., x_n$,

$$\begin{aligned} f(\vec{x}) &= \{F(p_1, ..., p_n) \mid p_1\colon N \to x_1, ..., p_n\colon N \to x_n, F \in I\} \\ &= \bigcup \{F(\vec{x}) \mid F \in I\}. \end{aligned}$$

7.2. *Reducibility in* BF. For each $p: N \to \mathsf{B}$ and $\pi: N \to N$, let

$$p^{\pi}(i) = p(\pi(i)),$$

and if $F \in \mathsf{BF}_2$ and $\pi, \rho: N \to N$, set

$$F^{\pi,\rho}(p, q) = F(p^{\pi}, q^{\rho}), (p, q: N \to \mathsf{B}). \tag{30}$$

We say that an infinitary behavior function G of two arguments is *reducible to F* and write $G \leqslant F$, if $G = F^{\pi,\rho}$ for suitable $\pi, \rho: N \to N$. The version of (30) for functions of one variable is

$$F^{\pi}(p) = F(p^{\pi}) = F(\lambda(i)\, p(\pi(i)))$$

and a bit messier for functions of n arguments.

Quite trivially, for all $p: N \to \mathsf{B}$, $\pi, \rho: N \to N$,

$$(p^{\pi})^{\rho} = p^{\pi\rho},$$
$$F^{\pi\rho}(p) = F(p^{\pi\rho}) = F^{\rho}(p^{\pi}),$$

where $\pi\rho$ is the composition of $\pi, \rho: N \to N$. Similar (messier) equations hold for functions with more arguments and we will use them routinely, without reference.

FACT 7.3. *The relation $\leqslant$ of reducibility is a (partial) preorder (transitive and reflexive) on each BF_n. Moreover,*

$$G \leqslant F \Rightarrow (\forall \vec{x})[G(\vec{x}) \subseteq F(\vec{x})],$$

so that (in particular) if F is an abstract implementation of some function $f: \mathscr{P}^n \to \mathscr{P}$, then so is G.

Proof. Suppose for simplicity that F, G are unary and $G = F^{\pi}$ for some $\pi: N \to N$. The inclusion $F^{\pi}(x) \subseteq F(x)$ follows trivially from the fact that $range(p) \subseteq x$ implies that $range(p^{\pi}) \subseteq x$. ∎

We now come to the basic definition of the modeling.

7.4. An *implemented player function* (*ipf*)[7] is a set f of infinitary behavior functions (f's *abstract implementations*) closed under reducibility. For each n-tuple of players $\vec{x} = x_1, ..., x_n$, the *player value* $f(\vec{x})$ is defined by

$$f(\vec{x}) = \{F(p_1, ..., p_n) \mid p_1: N \to x_1, ..., p_n: N \to x_n, F \in f\}.$$

[7] In Moschovakis (1989) we used the term *intensional player functions* (also abbreviated ipf) for this notion.

A set $I \subseteq \mathsf{BF}$ *generates* f if

$$f = \{F \mid F \leqslant G \text{ for some } G \in I\}.$$

Note that (immediately from 7.3), if I generates f, then it determines the values of f,

$$f(\vec{x}) = \bigcup \{F(\vec{x}) \mid F \in I\}.$$

The requirement of closure under reducibility ensures that ipf's view their arguments as *sets* rather than *sequences* of behaviors and eliminates trivial distinctions. Without it, $\{\lambda(p)\, p(0)\}$ and $\{\lambda(p)\, p(1)\}$ would be considered distinct intentional representations of the identity function, and it is not clear how that could be useful.

7.5. *Lifting behavior functions to ipf's.* With each (finitary) behavior function F: $\mathsf{B}^n \to \mathsf{B}$ we associate the ipf F^J generated by the single behavior function

$$F'(p_1, ..., p_n) = F(p_1(0), ..., p_n(0)).$$

When no ambiguity can arise we will identify F with its "lift-up" F^J.

For example, the ipf *next* modeling sequential composition of processes is the set of all behavior functions reducible to

$$S(p, q) = p(0) \textit{ next } q(0),$$

i.e. (easily) the set of all behavior functions of the form

$$S_{k,l}(p, q) = p(k) \textit{ next } q(l) \qquad (k, l \in N).$$

Similarly, the n-ary *projection function*

$$proj^{n,i}(x_1, ..., x_n) = x_i \qquad (1 \leqslant i \leqslant n)$$

is identified with the set of all $P_k^{n,i}$'s,

$$P_k^{n,i}(p_1, ..., p_n) = p_i(k). \tag{31}$$

When $n = i = 1$,

$$proj^{1,1} = id \tag{32}$$

is the ipf representation of the identity function on players, whose implementations are precisely all $P_k^{1,1}$'s. On the other hand, the ipf *twice*

which models the process function x *next* x is generated by the single behavior function

$$T(p) = p(0) \text{ } next \text{ } p(1), \tag{33}$$

and it is not the lift-up F^{J} of any behavior function F—that would have to be a unary F since *twice* is unary and T is unary and T is clearly not reducible to any unary F.[8]

7.6. *Intensional and extensional equality of ipf's.* Clearly $f = g$ implies the *extensional equality* $(\forall z)[f(z) = g(z)]$ but not vice versa.[9] We adopt an intensional use of λ, so that if a term $t(x)$ has a ckear intensional meaning, then $f = \lambda(x)\, t(x)$ is the ipf defined by it. We will also use on occasion the convenient abbreviation

$$f(z) =_{proc} g(z) \Leftrightarrow f = g.$$

7.7. *Composition of ipf's.* For given ipf's g, h_1, h_2, the composition

$$f = \lambda(x)\, g(h_1(x), h_2(x))$$

is the ipf generated by all behavior functions of the form

$$F(p) = G(\lambda(i)\, H_{1,i}(p), \lambda(i)\, H_{2,i}(p)),$$

[8] The abstract implementations in *twice* are all behavior functions of the form

$$T_{k,l}(p) = p(k) \text{ } next \text{ } p(l) \qquad (k, l \in N),$$

including the implementations $T_{k,k}$ which compute x *next* x by storing the behavior in x supplied by the first call to their argument and then using the same behavior for the second call, if one is needed. One may argue that it is not natural to admit these "smart" implementations in *twice* and it is possible to introduce finer, intensional modelings of process transformations which avoid it, e.g. by taking ipf's to be subsets of BF closed under the equivalence relation induced by the preorder $\leqslant$. Here we are primarily interested in studying the extensional properties of process transformations and we have adopted the coarsest intensional modeling for which we can develop the theory; see 9.6.

[9] For example, in the trivial structure with just the one (initial) state ι, the behavior function

$$I(p) = \begin{cases} \varnothing & \text{if } p(2)(\iota)\uparrow \text{ or } p(3)(\iota)\uparrow, \\ p(0), & \text{otherwise if } p(2)(\iota) \simeq p(3)(\iota), \\ p(1), & \text{otherwise} \end{cases}$$

is (easily) an implementation of the identity but it is clearly not reducible to any $P_k^{1,1}$ in (31); thus $P_0^{1,1}$ and $\{P_0^{1,1}, I\}$ generate distinct ipf's which are extensionally equal. Note that to compute $id(x)$ by this I we might have to make three distinct calls to x, and from a completely naive point of view it seems quite absurd to admit I is a "natural" implementation of *id*.

where G is an abstract implementation of g and $H_{1,i}$, $H_{2,i}$ are abstract implementations of h_1, h_2 respectively. The definition is similar for n-ary compositions.

FACT 7.8. (1) *If* $f = \lambda(x)\, g(h(x))$, *then for every* x, $f(x) = g(h(x))$, *and similarly for the composition of functions of any number of arguments.*

(2) *For each binary ipf* g, *if* $h = \lambda(x, y)\, g(proj^{2,1}(x, y), proj^{2,2}(x, y))$ *is the ipf obtained by composing* g *with the indicated projection functions, then* $h = g$, *i.e.,*

$$g(proj^{2,1}(x, y), proj^{2,2}(x, y)) =_{proc} g(x, y).$$

(*This is a sample result for compositions of this type.*)

Proof. To show first that $f(x) \subseteq g(h(x))$ in part (1), assume that $\sigma \in f(x)$, so by the definition

$$\sigma = G(\lambda(i)\, H_i(p)),$$

for some p such that $p[N] \subseteq x$. By 7.3, each $H_i(x) \subseteq h(x)$, so each $q(i) = H_i(p)) \in h(x)$ and hence $\sigma = G(q) \in g(h(x))$. The converse inclusion $g(h(x)) \subseteq f(x)$ is equally simple.

To check the inclusion $g \subseteq h = \lambda(x, y)\, g(proj^{2,1}(x, y), proj^{2,2}(x, y))$ in part (2), suppose G is some abstract implemenation of g and for each i, choose the abstract implementations

$$H_{1,i}(q, r) = q(i), \qquad H_{2,i}(q, r) = r(i)$$

of $proj^{2,1}$ and $proj^{2,2}$, respectively; now the behavior function

$$H(q, r) = G(\lambda(i)\, H_{1,i}(q, r), \lambda(i)\, H_{2,i}(q, r)) = G(q, r)$$

is an abstract implementation of h by the definition, and it is exactly G. Conversely, the typical abstract implementation of h is of the form

$$H(q, r) = G(\lambda(i)\, q(\pi(i)), \lambda(j)\, r(\rho(j))) = G(q^{\pi}, r^{\rho})$$

for suitable $\pi, \rho: N \to N$ and $G \in g$, so $H \in g$ by the closure of g under reducibility. ▌

The point of (2) is that we will be using (often without mention) the classical method of Gödel for reducing explicit definition to composition, e.g.,

$$f(x, y, x) = f(proj^{2,1}(x, y), proj^{2,2}(x, y), proj^{2,1}(x, y)). \tag{34}$$

We want to be sure that the intensional version of (34) is also valid, i.e.,

$$f(x, y, x) =_{proc} f(proj^{2,1}(x, y), proj^{2,2}(x, y), proj^{2,1}(x, y)).$$

8. Process Recursion

Suppose that f is a unary ipf and assume first that each abstract implementation $F \in f$ lives on a singleton; i.e., it depends only on one argument so that (in effect) $F\colon \mathsf{B} \to \mathsf{B}$. To compute the "least" fixed point of the equation $x = f(x)$ in the usual way, we want to start computing some $F_0 \in f$ on an unspecified argument and interpret each "call" for x as a call for some arbitrary $F \in f$. This suggests setting

$$\bar{x} = \{\lim_{n \to \infty} F_0 F_1 \cdots F_n(\varnothing) \mid F_0, F_1, \ldots \in f\}, \tag{35}$$

which in fact yields a "natural" fixed point of $x = f(x)$. We can reformulate this definition, by assigning to each sequence $F = (F_0, F_1, \ldots)$ of behavior functions in f the sequence $(\bar{F}_0, \bar{F}_1, \ldots)$ of simultaneous least fixed points of the equations

$$\sigma_i = F_i(\sigma_{i+1}) \qquad (i = 0, 1, \ldots),$$

and verifying that (35) is equivalent to

$$\bar{x} = \{\bar{F}_0 \mid F_0, F_1, \ldots \in f\}.$$

This form of the definition generalizes to the case where f is an arbitrary ipf, as follows.

8.1. *Simple fixed points.* An *implementation system* for a single equation

$$x = f(x)$$

is a set

$$\mathsf{F} = \{F_u \mid u \in N^*\}$$

of abstract implementations in f indexed by the finite sequences of integers. For each such F we define the system

$$\bar{\mathsf{F}} = \{\bar{F}_u \mid u \in N^*\}$$

of behaviors as *the simultaneous least fixed points* of the equations

$$\sigma_u = F_u(\sigma_{u0}, \sigma_{u1}, \ldots) \qquad (u \in N^*),$$

where $ui = (i) =$ the extension of the sequence u by i. Finally, we let

$$\bar{x} = \{\bar{F}_\varnothing \mid \mathsf{F} \text{ is an arbitrary impl. system for } x = f(x)\} \tag{36}$$

be *the player defined recursively by the equation* $x = f(x)$. Note that in the notation established by 2.1,

$$\bar{F}_u = \bigcup_n \bar{F}_u^{(n)},$$

where the stages $\bar{F}_u^{(n)}$ are defined by the recursion

$$\bar{F}_u^{(n+1)} = F_u(\bar{F}_{u0}^{(n)}, \bar{F}_{u1}^{(n)}, \ldots).$$

THEOREM 8.2. *For each unary ipf f, if* $\bar{x}$ *is defined by* (36), *then*

$$\bar{x} = f(\bar{x}).$$

Moreover, if for some type a,

$$f\colon \mathscr{P}(\mathsf{a}) \to \mathscr{P}(\mathsf{a})$$

in the sense that $x \in \mathscr{P}(\mathsf{a}) \Rightarrow f(x) \in \mathscr{P}(\mathsf{a})$, *then* $\bar{x} \in \mathscr{P}(\mathsf{a})$.

Proof. Check first (easily) that for each implementation system F for f and each sequence $u \in N^*$, the set

$$\mathsf{F}(u) = \{F_{u*v} \mid v \in N^*\}$$

is also an implementation system for f and it satisfies

$$\overline{F(u)_v} = \bar{F}_{u*v},$$

so that in particular, for each $u \in N^*$,

$$\bar{F}_u = \overline{F(u)_\varnothing} \in \bar{x}.$$

Thus

$$\sigma \in \bar{x} \Rightarrow \sigma = \bar{F}_\varnothing \text{ for some } \mathsf{F} = F_\varnothing(\lambda(i)\, \bar{F}_{(i)}) \in f(\bar{x}).$$

Conversely, if $\sigma \in f(\bar{x})$, then $\sigma = F_0(p)$ with some $F_0 \in f$ and for each $i \in N$, $p(i) = \bar{F}^i_\varnothing \in \bar{x}$ for some implementation system F^i of f. Now define F by

$$F_\varnothing = F_0, \qquad F_{(i)*u} = F^i_u,$$

and check (easily) that $\sigma = F_0(p) = \bar{F}_\varnothing$, so that $\sigma \in \bar{x}$.

For the second assertion about types, the hypothesis means that if F is any abstract implementation of f and $p[N] \subseteq \mathsf{a}$, then $F(p) \in \mathsf{a}$. Using the fact that $\varnothing \in \mathsf{a}$ to get the induction started, we prove from this that if $u \mapsto \bar{F}_u^{(n)}$ is the nth iterate in the simultaneous recursion which determines

$u \mapsto \bar{F}_u$ for some implementation system F, then $\bar{F}_u^{(n)} \in$ a; hence $\bar{F}_u = \bigcup \bar{F}_u^{(n)} \in$ a, since types are clearly losed under monotone unions. ▌

The point of the second assertion in the theorem is that we can establish safety properties of fixed points from assertions about preservation of safety properties by the defining equations. This should be true of every interpretation of recursion.

We can extend directly the definition above to systems of equations, with parameters, using systems indexed by sequences of tuples. As an example, we treat the case for a system with two equations and one parameter.

8.3. *Mutual recursion.* We let

$$(\{0, 1\} \times N)^* = \{(u, v) \mid u \in \langle\{0, 1\}^*, v \in N^*, |u| = |v|\}$$

be the set of finite sequences of pairs from $\{0, 1\}$ and N, which we view as pairs of finite sequences of the same length. An implementation system for a system of equations (in the parameter z)

$$x = f(x, y, z)$$

$$y = g(x, y, z)$$

is a pair of sets indexed by $(\{0, 1\} \times N)^*$,

$$\mathsf{F} = \{F_{u,v} \mid (u, v) \in (\{0, 1\} \times N)^*\}, \qquad \mathsf{G} = \{G_{u,v} \mid (u, v) \in (\{0, 1\} \times N)^*\},$$

where each $F_{u,v}$ and each $G_{u,v}$ are abstract implementations in f and g, respectively. Each such pair (F, G) determines the indexed sets of infinitary behavior functions $\bar{F}_{u,v}, \bar{G}_{u,v}: N \to \mathsf{B}$

$$\{\bar{F}_{u,v} \mid (u, v) \in (\{0, 1\} \times N)^*\}, \qquad \{\bar{G}_{u,v} \mid (u, v) \in (\{0, 1\} \times N)^*\}$$

which are the simultaneous least fixed points of the equations

$$\sigma_{u,v}(r) = F_{u,v}(\lambda(i)\, \sigma_{u0,vi}(r), \lambda(j)\, \tau_{u0,vj}(r), r)$$

$$\tau_{u,v}(r) = G_{u,v}(\lambda(i)\, \sigma_{u1,vi}(r), \lambda(j)\, \tau_{u1,vj}(r), r),$$

and the implemented player functions defined recursively by the system are

$$\bar{x} = \{\bar{F}_{\varnothing,\varnothing} \mid (\mathsf{F}, \mathsf{G}) \text{ as above}\}, \qquad \bar{y} = \{\bar{G}_{\varnothing,\varnothing} \mid (\mathsf{F}, \mathsf{G}) \text{ as above}\}.$$

The motivation is exactly as in the case of a single equation; i.e., we think of computing $\bar{x}(z)$ by starting on $f(x, y, z)$ on the given z and unspecified x, y and "calling" f when x is required and g when y is required; these "calls" must be via totally independent abstract implementations of f and

g, $F_{u,v}$ and $G_{u,v}$, where the indexing is coded so that when u ends in 0 the call is by f and when the u ends in 1 then the call is by g. For example, with $u = v = \varnothing$ and skipping the argument r, i.e., abbreviating

$$\bar{F}_{u,v} = \bar{F}_{u,v}(r), \qquad \bar{G}_{u,v} = \bar{G}_{u,v}(r)$$

to simplify the notation on the right,

$$\bar{F}_{\varnothing,\varnothing}(r) = F_{\varnothing,\varnothing}(\bar{F}_{(0),(0)}, \bar{F}_{(0),(1)}, \bar{F}_{(0),(2)}, ..., \bar{G}_{(0),(0)}, \bar{G}_{(0),(1)}, \bar{G}_{(0),(2)}, ..., r)$$
$$\bar{G}_{\varnothing,\varnothing}(r) = G_{\varnothing,\varnothing}(\bar{F}_{(1),(0)}, \bar{F}_{(1),(1)}, \bar{F}_{(1),(2)}, ..., \bar{G}_{(1),(0)}, \bar{G}_{(1),(1)}, \bar{G}_{(1),(2)}, ..., r).$$

For the general case of systems of n equations the definition uses systems indexed by $(\{0, ..., n-1\} \times N)^*$.

THEOREM 8.4. *The ipf's defined recursively by a system satisfy the system; thus for the special case of a system of two equations as in* 8.3,

$$\bar{x}(z) =_{proc} f(\bar{x}(z), \bar{y}(z), z),$$
$$\bar{y}(z) =_{proc} g(\bar{x}(z), \bar{y}(z), z).$$

Moreover, if a, b, *are types and*

$$f: \mathscr{P}(\mathsf{a}) \times \mathscr{P}(\mathsf{b}) \times \mathscr{P}(\mathsf{c}) \to \mathscr{P}(\mathsf{a}),$$
$$g: \mathscr{P}(\mathsf{a}) \times \mathscr{P}(\mathsf{b}) \times \mathscr{P}(\mathsf{c}) \to \mathscr{P}(\mathsf{b}),$$

then $\bar{x}: \mathscr{P}(\mathsf{c}) \to \mathscr{P}(\mathsf{a})$, $\bar{y}: \mathscr{P}(\mathsf{c}) \to \mathscr{P}(\mathsf{b})$.

Proof is an elaboration of the proof of 8.2, so we will only only outline briefly the

$$\lambda(z)\, f(\bar{x}(z), \bar{y}(z), z) \subseteq \bar{x} \tag{37}$$

part which involves composition of ipf's. Replacing z by $id(z)$ to reduce the explicit definition to composition and using 7.8, we get that the arbitrary abstract implementation of the left-hand-side of (37) is of the form

$$L(r) = F'(\lambda(i)\, X_i(r), \lambda(i)\, Y_i(r), \lambda(i)\, r(\pi(i))),$$

where $F' \in f$, each X_i and Y_i are in $\bar{x}$, $\bar{y}$, respectively, and $\pi: N \to N$. By the definition of the fixed points, we can find systems of implementations $(\mathsf{F}^{0,i}, \mathsf{G}^{1,i})$ and $(\mathsf{F}^{1,i}, \mathsf{G}^{0,i})$ (note the asymmetric indexing) which define X_i and Y_i respectively, so that

$$L(r) = F(\lambda(i)\, \overline{F^{0,i}}_{\varnothing,\varnothing}(r), \lambda(i)\, \overline{G^{0,i}}_{\varnothing,\varnothing}(r), r),$$

where we have also replaced F' by F on the simpler argument r using the closure of f under reducibility. As in the proof of 8.2 now, define the pair (F, G) as follows:

$$F_{\emptyset,\emptyset} = F, \qquad G_{\emptyset,\emptyset} = \text{any } G \in g,$$
$$F_{iu,jv} = F^{i,j}_{u,v}, \qquad G_{iu,jv} = G^{i,j}_{u,v}.$$

It follows easily that $\bar{F}_{iu,jv} = \overline{F^{i,j}}_{u,v}$, $\bar{G}_{iu,jv} = \overline{G^{i,j}}_{u,v}$, so that in particular

$$\begin{aligned}\bar{F}_{\emptyset,\emptyset}(r) &= F(\lambda(i)\,\bar{F}_{(0),(i)}(r), \lambda(i)\,\bar{G}_{(0),(i)}(r), r)\\ &= F(\lambda(i)\,\overline{F^{0,i}}_{\emptyset,\emptyset}(r)\,\lambda(i)\,\overline{G^{0,i}}_{\emptyset,\emptyset}(r), r)\\ &= L(r)\end{aligned}$$

and $L = \bar{F}_{\emptyset,\emptyset} \in \bar{x}$. ▮

There is no obvious characterization of these fixed points in terms of some order in $\mathscr{P}$, because (as is well understood) spaces of sets like $\mathscr{P}$ do not carry nice orders. However:

FACT 8.5. *The fixed point $\bar{x}$ of $x = f(x)$ defined by* 8.1 *is minimal among fixed points of $x = f(x)$ in the upper preorder*[10] *on players; i.e., for every fixed point $\bar{y}$ of $x = f(x)$,*

$$(\forall \tau \in \bar{y})(\exists \sigma \in \bar{x})[\sigma \subseteq \tau].$$

It follows that if $\bar{x}$ is a total player, then it is the largest (as a set) fixed point of f.

Proof. Assuming that $\tau \in \bar{y}$ and $f(y) = y$, we have that $\tau = F_{\emptyset}(\lambda(i)\tau_i)$ for some $F_{\emptyset} \in f$ and suitable $\tau_i \in \bar{y}$, so that in turn, for each i

$$\tau_i = F_{(i)}(\lambda(j)\,\tau_{i,j})$$

for some $F_{(i)} \in f$ and $\tau_{i,j} \in \bar{y}$, etc., so that we can choose an implementation system $\mathsf{F} = \{F_u \mid u \in N^*\}$ for f and members $\tau_u \in \bar{y}$ satisfying

$$\tau_u = F_u(\lambda(i)\,\tau_{u*(i)}) \qquad (u \in N^*), \tag{38}$$

with the given $\tau = \tau_{\emptyset}$. Since the behaviors $\{\bar{F}_u \mid u \in N^*\}$ in $\bar{x}$ determined by F are the least simultaneous fixed points of the equations (38), we have $\bar{F}_u \subseteq \tau_u$ for every u, and in particular this holds for $u = \emptyset$, completing the proof of the main assertion. The corollary follows immediately. ▮

[10] This is a natural ppreorder on classes of subsets of a poser used in the construction of powerdomains. In Moschovakis (1989) we called it the "Milner preorder," following Broy (1986), but it appears that "upper" is a more appropriate name for it.

This is the only obvious restriction on $\bar{x}$ in terms of natural preorders on $\mathscr{P}$.

8.6. EXAMPLES. Fix the trivial structure with just the one state ι and acts a, b, and *skip*, so that behaviors are just streams of these acts and behavior functions are monotone, continuous stream functions.

(1) The equation

$$x = (a\ next\ x)\ or\ (b\ next\ x).$$

has no $\subseteq$-least solution, since the two sets of infinite sequences of a's and b's which are ultimately constant both satisfy it; $\bar{x}$ is the $\subseteq$-largest fixed point, which is also the unique upper-minimal fixed point.

(2) For the most trivial equation $x = x$, $\bar{x} = \{\varnothing\}$ is the $\subseteq$-least fixed point.

(3) The equation

$$x = x\ or\ b\ or\ (a\ next\ x)$$

has a $\subseteq$-least fixed point which consists of all the finite, convergent streams of the form $a \cdots ab1$; it has a $\subseteq$-largest fixed point which consists of all streams; it has a $\subseteq$-least, upper-minimal fixed point which consists of all the finite divergent streams of a's and all the streams in the $\subseteq$-least fixed point; and $\bar{x}$ is none of these: it consists of all the streams in the $\subseteq$-least, upper-minimal fixed point together with the infinite stream $a \cdot a \cdots$. Note that the streams in $\bar{x}$ are what we would expect to get by implementing the "naive" understanding of this equation.

It is quite easy to construct more complex examples which make it seem quite unlikely that we can find a "structural" (algebraic) characterization of this notion of recursion. Thus we should look for its justification on the general properties it satisfies, and these are expressed naturally as results about the semantics of $\mathscr{L}$-structures where we take ipf's as the given functions.

8.7. A *process structure* is a pair $\mathscr{A} = (\mathscr{S}, \mathscr{F})$, where $\mathscr{S}$ is a state structure and $\mathscr{F}$ assigns an n-ary ipf $\mathscr{F}(f)$ to each function symbol of arity n. In the *process semantics* for $\mathscr{L}$ we associate with each expression E and each n-tuple of variables $\vec{x}$ which includes all the free variables of E an ipf

$$f_E = process(\mathscr{A}, \vec{x})E,$$

using ipf composition defined by 7.7 and interpreting recursive expressions using ipf recursion: e.g., if

$$E \equiv \mathrm{rec}(u, v)[E_0(u, v), E_1(u, v), E_2(u, v)],$$

then for any $\vec{x}$ which includes all the free variables of E we let

$$f_i = process(\mathscr{A}, u, v, \vec{x}) E_i \qquad (i = 0, 1, 2),$$

we let $\bar{u}$, $\bar{v}$ be the ipf's defined recursively by the system

$$u = f_1(u, v, \vec{x})$$
$$v = f_2(u, v, \vec{x}),$$

and we set

$$process(\mathscr{A}, \vec{x}) E = \lambda(\vec{x}) f_0(\bar{u}(\vec{x}), \bar{v}(\vec{x}), \vec{x}).$$

If E is a *closed expression* (with no free variables), then its process denotation f_E relative to the empty list is an ipf with no arguments, i.e., just a player. Two expressions E, M are *process equivalent* if they have the same process denotation $f_E = f_M$ on every process structure and for every $\vec{x}$. We write

$$E \equiv_{proc} M \Leftrightarrow E, M \text{ are process equivalent.}$$

8.8. *Lifting behavior into process semantics.* For each behavior σ in a behavior structure $\mathscr{L}$, set

$$\jmath(\sigma) = \{\sigma\},$$

and with each behavior function F (binary, for simplicity), associate the process function $F^{\jmath}$ defined by 7.5, whose abstract implementations are all behavior functions of the form

$$F^{k,l}(p, q) = F(p(k), q(l)).$$

The next result expresses the faithfulness of this interpretation and it can be proved by a direct fixed-point argument which we will skip.

THEOREM 8.9. *Let $\mathscr{A}^{\jmath}$ be the process structure (of the same signature) associated with a behavior structure $\mathscr{A}$ by replacing each behavior function F in $\mathscr{A}$ by the corresponding process function $F^{\jmath}$. For every expression E, then, if F_E and f_E are the functions associated with E in $\mathscr{A}$ and $\mathscr{A}^{\jmath}$, respectively, then for all behaviors $\sigma_1, \ldots, \sigma_n$,*

$$F_E(\sigma_1, \ldots, \sigma_n) = \jmath^{-1} f_E(\jmath(\sigma_1), \ldots, \jmath(\sigma_n)).$$

9. THE MAIN PROPERTIES OF THE MODEL

In the next section we will prove the following main result of the paper.

THEOREM 9.1. The Transfer Principle. *If two expressions of the language $\mathscr{L}$ are behavior equivalent, then they are also process equivalent; in symbols,*

$$E \equiv_{beh} M \Rightarrow E \equiv_{proc} M.$$

The converse of this is immediate from 8.9, *so that the expression identities which are valid in all behavior structure are also valid in all process structures.*

Since recursion is defined by the taking of least fixed points in behavior semantics, we have immediately the corollary:

THEOREM 9.2. *Least-fixed-point equivalent $\mathscr{L}$-expressions are also process equivalent.*[11]

The fact that fixed points satisfy their defining equations is an obvious special case of this result, since

$$f(\mathrm{rec}(x)[x, f(x)]) = \mathrm{rec}(x)[x, f(x)]$$

is valid in least-fixed-point semantics so that by 9.2 it also holds for process semantics. This is Theorem 8.2, a special case of the more general 8.4, which also follows from 9.2. Additional basic corollaries are the well-known rules which reduce nested to mutual recursion:

THEOREM 9.3 The Bekič–Scott Rules. *For all expressions and sequences of expressions and variables as indicated, the following identities are valid in process semantics:*

$$\mathrm{rec}(\vec{z})[\mathrm{rec}(\vec{y})[E, \vec{M}], \vec{N}] = \mathrm{rec}(\vec{y}, \vec{z})[E, \vec{M}, \vec{N}], \tag{39}$$

$$\mathrm{rec}(\vec{x}_1, z, \vec{x}_2)[\vec{E}_1, \mathrm{rec}(\vec{y})[M, \vec{N}], \vec{E}_2] = \mathrm{rec}(\vec{x}_1, z, \vec{y}, \vec{x}_2)[\vec{E}_1, M, \vec{N}, \vec{E}_2]. \tag{40}$$

In fact we will need to prove these first, as lemmas for the general transfer

[11] [Added November 29, 1990] I now have a complete axiomatization of least-fixed-point equivalence for expressions of $\mathscr{L}$ and a proof of the decidability of this class of identities. Using this construction, Tonny Hurkens has proved that *least-fixed-point equivalence concides with procedure equivalence for expressions of $\mathscr{L}$*, and hence also with process equivalence by the results of this paper.

principle. Perhaps they are easier to recognize in the following form which refers directly to ipf recursion.

THEOREM 9.4. *Suppose the ipf's on the right in the list below are defined by the recursions on the left*:

$$x = f(x, y, z): \qquad \check{x} = \lambda(y, z)\, \check{x}(y, z)$$

$$y = g(\check{x}(y, z), y, z): \check{y} = \lambda(z)\, \check{y}(z)$$

$$\left.\begin{array}{l} x = f(x, y, z) \\ y = g(x, y, z) \end{array}\right\}: \qquad \left.\begin{array}{l} \bar{x} = \lambda(z)\, \bar{x}(z) \\ \bar{y} = \lambda(z)\, \bar{y}(z) \end{array}\right\}.$$

We then have

$$\bar{y} = \check{y},$$

$$\bar{x} = \lambda(z)\, \check{x}(\bar{y}(z), z),$$

and hence,

$$(\forall z)[\bar{x}(z) = \check{x}(\bar{y}(z), z)].$$

9.5. *Limitations of the theorem.* Suppose $\mathscr{L}'$ is the extension of $\mathscr{L}$ by a binary construct $\mathsf{cond}(x, y)$ and we agree that in the behavior semantics of $\mathscr{L}'$, $\mathsf{cond}(x, y)$ is to be interpreted by a conditional behavior function $COND_R$ based on a relation R on the states

$$COND_R(\sigma, \tau) = \lambda(\bar{s}(n))[\text{if } R(s_0) \text{ then } \sigma(\bar{s}(n)) \text{ else } \tau(\bar{s}(n))]; \tag{41}$$

in the process semantics of $\mathscr{L}'$, $\mathsf{cond}(x, y)$ is to be interpreted by an ipf generated by some behavior conditional satisfying (41). Now the identity

$$\mathsf{cond}(x, x) = x \tag{42}$$

is valid in all behavior structures, simply because for every s_0, $R(s_0)$ is either true or false, but it is not valid in process structures because if $x = \{a, b\}$, then the right-hand-side of (42) has only the two, trivial behaviors a and b, while the left-hand-side has the additional state-dependent behavior

$$\rho(\bar{s}(n)) = \text{if } R(s_0) \text{ then } a \text{ else } b.$$

Thus *we cannot extend the transfer principle directly to languages with conditionals.*

This simple obstruction has nothing to do with recursion, it depends instead on the well-understood "algebraic" difficulties that come up when

we try to use conditionals in a non-deterministic setting. Although it forbids direct extensions of the transfer principle to languages even minimally richer than $\mathscr{L}$, this example still allows for weaker transfer principles with which it is consistent. For example, the two sides of (42) are "observationally equivalent" under any reasonable definition of "observation" for processes: perhaps one can show that behavior equivalent expressions in fairly rich languages are observationally equivalent in the appropriate process semantics, with a natural, rigorous definition of "observational equivalence."

9.6. *Extensional process semantics.* A function $f: \mathscr{P}^n \to \mathscr{P}$ on the set of players is (abstractly) *implementable* if *some* set I of its abstract implementations determines it, so that it is extensionally equal to the ipf $f^* = AI(f)$ of all its abstract implementations. An *extensional process structure* is one where $\mathscr{F}$ assigns implementable player functions to the function symbols and we can define semantics for these structures by identifying every implementable function f with the ipf $AI(f)$ and using our definition of recursion. The problem is that we cannot prove the basic Bekič–Scott Rules 9.3 for this semantics because we have not been able to solve the following basic problem about process recursion:

9.7. *Question.* Suppose f and g are extensionally equal implemented player functions and the equations $x = f(x)$, $x = g(x)$ determine the fixed points $\bar{x}_f$ and $\bar{x}_g$ respectively; must we have $\bar{x}_f = \bar{x}_g$?

It appears quite difficult to answer this question positively with the methods we will use to prove 9.1. A positive answer would be foundationally interesting, as it would provide a good, extensional model combining fair merge with full recursion, but it is more likely that the answer is negative. Of course, there may be an enitrely different way to approach the subject with some other, natural notion of process recursion which bypasses the question entirely.

10. APPENDIX. Proof of the Transfer Principle

It is convenient for the proofs to enrich the language $\mathscr{L}$ with the constant (0-ary function symbol) $\perp$ which abbreviates the recursive expression

$$\perp \equiv \mathrm{rec}(x)[x, x]. \tag{43}$$

This is interpreted in behavior structures by the totally undefined behavior $\varnothing$ and in process structures by the deterministic player $\{\varnothing\}$. We will need

some preliminary definitions and lemmas, beginning with a "parametrization" of the notion of abstract implementation which has more structure and is easier to compute with.

10.1. *A representation of an n-ary ipf f* is any function

$$F: N \times (N \to \mathsf{B})^n \to \mathsf{B}, \tag{44}$$

such that for each $u \in N$, $\lambda(\vec{p})\, F(u, \vec{p}) \in f$, i.e., the function F_u defined by

$$F_u(\vec{p}) = F(u, \vec{p})$$

is an abstract implementation of f. A representation of an $\mathscr{L}$-expression E is a representation of the ipf f_E defined by E—all relative to some structure $\mathscr{A}$ and some list $\vec{x}$ of variables including all the free variables of E, which we will typically leave implicit.

By "representation" we will mean a representation of *some* f, and this comes down to just a monotone, continuous function as in (44). A representation F' is *reducible* to another F if there exists some $\pi: N \to N$ such that

$$F'(u, \vec{p}) = F(\pi(u), \vec{p}).$$

In this case, if F is a representation of some ipf f or some expression E, then so is F'. For example, the representations of the identity ipf $\lambda(x)x$ are all functions of the form $F_\pi(u, p) = p(\pi(u)) = p^\pi(u)$, where $\pi: N \to N$, and there is only one representation of the constant $\perp$, the constant function $\lambda(u)\varnothing$.

To deal with least-fixed-point recursive definitions of representations, we will use informally some obvious extensions of the recursion construct of $\mathscr{L}$. For example,

$$\mathtt{rec}(u, p, v, q)[F(i, p, q), G(u, p, q), H(v, p, q)] = F(i, \bar{p}, \bar{q}),$$

where $\bar{p}, \bar{q}$ are the simultaneous least-fixed-points of the system

$$p(u) = G(u, p, q)$$
$$q(v) = H(v, p, q).$$

Note that this system of equations is just the "point form" of

$$p = \lambda(u)\, G(u, p, q)$$
$$q = \lambda(v)\, H(v, p, q).$$

We will use interchangeably the two notations

$$\lambda(i)\,\mathtt{rec}(u, p, v, q)[F(i, p, q), G(u, p, q), H(v, p, q)]$$
$$= \mathtt{rec}(p, q)[\lambda(i)\,F(i, p, q), \lambda(u)\,G(u, p, q), \lambda(v)\,H(v, p, q)].$$

10.2. Representation Lemma. *If the ipf h is defined by the recursion*

$$h(\vec{z}) = \mathtt{rec}(\vec{x})[o(\vec{x}, \vec{z}), f_1(\vec{x}, \vec{z}), ..., f_n(\vec{x}, \vec{z})],$$

then its representations are exactly all functions H satisfying

$$H(i, \vec{r}) = \mathtt{rec}(u_1, p_1, ..., u_n, p_n)[O(i, \vec{p}, \vec{r}), F_1(u_1, \vec{p}, \vec{r}), ..., F_n(u_n, \vec{p}, \vec{r})],$$

where O and $F_1, ..., F_n$ are representations of o and $f_1, ..., f_n$ respectively.

Proof. Assume for simplcity that the recursion is not mutual ($n = 1$) and $\vec{z} \equiv z$ is a single variable, i.e.,

$$h(z) = \mathtt{rec}(x)[o(x, z), f(x, z)].$$

Part 1.[12] Every representation H of h satisfies

$$H(u, r) = \mathtt{rec}(i, p)[O(u, p, r), F(i, p, r)] \tag{45}$$

with suitably chosen representations O and F of o and f, respectively.

By the definition of composition and ipf recursion, each representation of h is of the form

$$H(u, r) = O'_u(\lambda(i)\,X'_{u,i}(r), \lambda(j)\,r(\pi_u(j)))$$
$$= O(u, \lambda(i)\,X_{u,i}(r), r), \tag{46}$$

where each $\pi_u: N \to N$ and in the second equation O is a representation of o and each $X_{u,i}$ is an implementation of the ipf $\bar{x}$ defined by the recursion $x = f(x, z)$; we have used here—and in the sequel will use without reference—the closure properties of classes of representations. By the definition of ipf recursion now, $X_{u,i} = \bar{F}_{u,i,\varnothing}$ $(u, i \in N)$, with suitably chosen implementation systems $\mathsf{F}_{u,i} = \{F_{u,i,v} \mid v \in N^*\}$ $(u, i \in N)$. Switching to single-integer indexing, set for each $j \in N$

$$\mathsf{F}_j = \mathsf{F}_{(j)_0, (j)_1}$$
$$X_j(r) = X_{(j)_0, (j)_1} = \bar{F}_{j,\varnothing}(r).$$

[12] In the computations of this section we use repeatedly a fixed coding of integer tuples $\langle\ \rangle: N^* \xrightarrow{1-1} N$ with inverse $(w, i) \mapsto (w)_i$, so that, e.g., $(\langle n, m\rangle)_0 = n$, $(\langle n, m\rangle)_1 = m$. We also use the "algebraic" notation for λ-abstraction, $f(x, \cdot) = \lambda(y)\,f(x, y)$, so that, in particular, $\langle i, \cdot\rangle = \lambda(j)\langle i, j\rangle$.

Let $\pi: N \to N^*$ be an enumeration of all tuples without repetitions and such that $\pi(0) = \varnothing$ and set

$$F(j, i, p, r) = F_{j, \pi(i)}(\lambda(m)\, p(\langle j, \pi^{-1}(\pi(i) * (m))\rangle), r).$$

Fix r for the argument and let $\bar{p}$ be the least-fixed-point of the equation

$$\begin{aligned} p(s) &= F((s)_0, (s)_1, p, r) \\ &= F_{(s)_0, \pi((s)_1)}(\lambda(m)\, p(\langle (s)_0, \pi^{-1}(\pi((s)_1) * (m))\rangle), r). \end{aligned}$$

We claim that (for the fixed r),

$$X_j(r) = \bar{p}(\langle j, 0\rangle), \tag{47}$$

so that going back to (46) and computing, we have

$$\begin{aligned} H(u, r) &= O(u, \lambda(i)\, X_{u,i}(r)) && \text{by (46)}, \\ &= O(u, \lambda(i)\, X_{\langle u,i\rangle}(r), r) && \text{by def.}, \\ &= O(u, \lambda(i)\, \bar{p}(\langle\langle u, i\rangle, 0\rangle), r) && \text{by (47)}, \\ &= \mathtt{rec}(s, p)[O(u, \lambda(i)\, p(\langle\langle u, i\rangle, 0\rangle), r), F((s)_0, (s)_1, p, r)]. \end{aligned}$$

This will complete the proof of Part 1, because the expressions involving O and F are representations of o and f respectively, and this puts H in the required form (45).

To prove the claimed (47) we will show that (for the fixed r and all j, k),

$$\bar{F}_{j, \pi(k)}(k) = \bar{p}(\langle j, k\rangle),$$

from which (47) follows setting $k = 0$. The proof is by induction on the stage of the recursions which define $\bar{F}_{j, \pi(k)}(r)$ and $\bar{p}$, i.e., we will show that for each n,

$$\bar{F}^{(n)}_{j, \pi(k)}(r) = \bar{p}^{(n)}(\langle j, k\rangle).$$

At the induction step we have

$$\begin{aligned} \bar{F}^{(n+1)}_{j, \pi(k)} &= F_{j, \pi(k)}(\lambda(m)\, \bar{F}^{(n)}_{j, \pi(k) * (m)}, r) && \text{by def.} \\ &= F_{j, \pi(k)}(\lambda(m)\, \bar{p}^{(n)}(\langle j, \pi^{-1}(\pi(k) * (m))\rangle), r) && \text{by ind. hyp.} \\ &= \bar{p}^{(n+1)}(\langle j, k\rangle) && \text{by def.} \end{aligned}$$

Part 2. Every H which satisfies (45) is a representation of the ipf h. Given (45), define the system G for the equation $x = f(x, z)$ by

$$G_{\varnothing}(p, r) = G_{(0)}(p, r) = F(0, p, r)$$
$$G_{vi}(p, r) = F(i, p, r)$$

which is quite degenerate in that G_v depends only on the last term in the sequence $v \in N^*$. The proof uses the fact that for this (and every) implementation system for f, each of the simultaneous least-fixed-points $\bar{G}_v$ ($v \in N^*$) is an abstract implementation of the ipf $\bar{x}$ defined by the recursion $x = f(x, z)$, which is easy to check as in the proof of 8.2.

Fix r and let $\bar{p}$ be the least-fixed-point of

$$p(i) = F(i, p, r);$$

we will check by induction on the stage n of the recursions defining $\bar{G}_v$ and $\bar{p}$ that

$$\bar{G}_{vi}^{(n)}(r) = \bar{p}^{(n)}(i), \tag{48}$$

so that in the limit $\bar{p}(i) = \bar{G}_{(i)}(r)$, and by the observation above each $\bar{p}(i)$ is in $\bar{x}$. From this it follows immediately that $H(u, r) = O(u, \bar{p}, r)$ is a representation of h. Finally, for the induction step in the proof of (48) we compute

$$\begin{aligned} \bar{G}_{vi}^{(n+1)} &= G_{vi}(\lambda(m)\, \bar{G}_{v*(i,m)}^{(n)}(r), r) && \text{by def.} \\ &= F(i, \bar{p}^{(n)}, r) && \text{by ind. hyp. and def. of } G_{vi}, \\ &= \bar{p}^{(n+1)}(i) && \text{by def.} \end{aligned}$$

This completes the proof of Part 2 and the proof of the Lemma. ∎

10.3. LEMMA. *For all ipf's* $f, g_1, ..., g_n$,

$$f(g_1(\vec{x}), ..., g_n(\vec{x})) =_{proc} \mathtt{rec}(y_1, ..., y_n)[f(y_1, ..., y_n), g_1(\vec{x}), ..., g_n(\vec{x})]. \tag{49}$$

Proof. By the definition of composition (and with $n = 2$ for simplicity), the typical representation of the left-hand-side of (49) is of the form

$$\begin{aligned} LHS(u, r) &= F(u, \lambda(i)\, G_1(i, r), \lambda(j)\, G_2(j, r)) \\ &= \mathtt{rec}(i, p, j, q)[F(u, p, q), G_1(i, r), G_2(j, r)], \end{aligned}$$

where F, G_1, G_2 are representations of f, g_1, g_2, respectively, and the

second equation follows from least-fixed-point recursion; now 10.2 implies that it is also a representation of the right-hand-side of (49). Proof of the converse inclusion is almost identical. ▮

10.4. *Proof of the Bekič–Scott Rules*, 9.3. The proof of (39) is very similar to the proof just given, using the fact that (39) is valid in least-fixed-point semantics. Beginning the same way for a proof of the special case

$$\mathtt{rec}(x)[E, \mathtt{rec}(y)[M, N]] = \mathtt{rec}(x, y)[E, M, N] \tag{50}$$

of (40), we know by 10.2 that the typical representation of the left-hand-side of (50) satisfies

$$L(u, r) = \mathtt{rec}(i, p)[F(u, p), \mathtt{rec}(j, q)[G(i, p, q), H(i, p, j, q)]]$$

where $F. G, H$ are representations of the ipf's defined by the respective expressions E, M, N. Now we can use the relevant Bekič–Scott rule for this kind of nested recursion (e.g., see Moschovakis, 1989a, Theorem 2B.4) to get

$$L(u, r) = \mathtt{rec}(i, p, i', j, r)[F(u, p), G(i, p, r(i, \cdot)), H(i', p, j, r(i', \cdot))].$$

This means that for each fixed r,

$$L(u, r) = F(u, \bar{p}),$$

where $\bar{p}$ is the first of the simultaneous least-fixed-points of the system

$$p(i) = G(i, p, r(i, \cdot))$$
$$r(i', j) = H(i', p, j, r(', \cdot)).$$

By contraction of variables, this system is (easily) equivalent to

$$p(i) = G(i, p, q(\langle i, \cdot \rangle))$$
$$q(s) = H((s)_0, p, (s_1, r(\langle (s)_0, \cdot \rangle))$$

in the sense that it defines the same first least-fixed-point $\bar{p}$. Thus we have

$$L(u, r) = \mathtt{rec}(i, p, s, q)[F(u, p), G(i, p, q(\langle i, \cdot) \rangle)),$$
$$H((s)_0, p, (s)_1, q(\langle (s)_0, \cdot \rangle))]$$

which by the closure properties of representations and 10.2 implies again that L is also a representation of the right-hand-side of (50). The converse is a bit simpler. ▮

In addition to these two basic lemmas, we also need to deal with identification of variables within recursion. Here is a typical result which can be proved easily using 10.2 and simple least-fixed-point arguments.

10.5. LEMMA. *The following identities are valid in both behavior and process semantics*:

$$\mathtt{rec}(x, y, z)[o(x, y, z), y, f(x, y, z), g(x, y, z)]$$
$$=_{proc} \mathtt{rec}(x, z)[o(x, x, z), f(x, x, z), g(x, x, z)]$$
$$\mathtt{rec}(x, y, z)[o(x, y, z), x, f(x, y, z), g(x, y, z)]$$
$$=_{proc} \mathtt{rec}(x, y, z)[o(x, y, z)[o(x, y, z), \bot, f(x, y, z), g(x, y, z)].$$

In the sequel we will use the validity of specific identities like this without comment.

10.6. *A simple expression* is one of the form $f(x_1, ..., x_n)$, where f is a function symbol and $x_1, ..., x_n$ are (not necessarily distinct) variables, e.g., $f(x, y, x, x)$. An expression E^* is *in simplified form* if

$$E^* \equiv \mathtt{rec}(x_1, ..., x_n)[E_0, E_1, ..., E_n], \tag{51}$$

where for $i = 1, ..., n$, E_i is simple and E_0 is either simple or a variable; we allow $n = 0$, in which case $E^* \equiv \mathtt{rec}(\ )[E_0]$ with E_0 simple or a variable. Note that this allows $E_i \equiv f(\ )$ for a 0-ary function symbol, including $\bot$.

10.7. SIMPLIFIED FORM LEMMA. *Each $\mathscr{L}$-expression E is both behavior and process equivalent to some expression E^* which is in simplified form.*

Proof. Using 9.3 and 10.3, show first by an easy induction on the length of E that we can find an equivalent E^* as in (51) with each E_i simple or a variable. Now apply 10.5 to eliminate the E_i's with $i \geqslant 1$ which are single variables. ∎

We have almost completed the preliminary work for the proof of the Transfer Principle except for one remaining difficulty. Suppose

$$E \equiv \mathtt{rec}(x, y)[f(x, y), g(x, y), f(x, y)]$$

and E is behavior equivalent to some expression M, also in simplified form. We know from 10.2 that for every representation H of E on a fixed process structure $\mathscr{A}$, there are representations F_1, F_2 of f and G of g such that

$$H = \mathtt{rec}(p, q)[\lambda(i)\, F_1(i, p, q), \lambda(j)\, G(j, p, q), \lambda(i)\, F_2(i, p, q)].$$

If $F_1 = F_2$, then H is defined by a formula which is essentially an alphabetic

variant of E, and it seems quite plausible that we can then concoct some behavior structure $\mathscr{A}'$ in which H will in fact be the behavior denotation of E, from which point it should be easy to complete the proof. The next lemma which guarantees that we can arrange for this equality $F_1 = F_2$ is the heart of the proof of the Transfer Principle.

10.8. AMALGAMATION LEMMA. *Suppose*

$$E \equiv \mathrm{rec}(x_1, ..., x_n)[E_0, E_1, ..., E_n]$$

is an $\mathscr{L}$-expression in simplified form, where in addition to the distinct variables $x_1, ..., x_n$ there may occur in E other distict variables $y_1, ..., y_k$ and distinct function symbols $f_1, ..., f_m$. Let $u_0, u_1, ..., u_n, p_1, ..., p_n, q_1, ..., q_k, F_1, ..., F_m$ be distinct symbols, set

$$x_m^0 \equiv p_m, \qquad y_m^0 \equiv q_m,$$

and put

$$E^0 \equiv \mathrm{rec}(u_1, p_1, ..., u_n, p_n)[E_0^0, E_1^0, ..., E_n^0],$$

where each E_i^0 is defined as follows:

1. *If $E_0 \equiv z_j$ is a variable, then $E_0^0 \equiv z_j^0$.*
2. *For any i, if $E_i \equiv f_j(z_1, ..., z_l)$, then $E_i^0 \equiv \lambda(u_i)\, F_j(u_i, z_1^0, ..., z_l^0)$.*

For each process structure $\mathscr{A}$, each representation of E is reducible to one definable by E^0, with a suitable choice of representations $F_1, ..., F_m$ of (the ipf's interpreting in $\mathscr{A}$) $f_1, ..., f_m$ respectively.

Conversely, every representation defined by E^0 in this way is a representation of E.

Proof. For example, suppose

$$E \equiv \mathrm{rec}(x, y, z)[o(x, x, y, w), f(x, w, z, y), f(y, y, w, z), o(w, w, x, y)], \tag{52}$$

so that the only free variable of E is w. The lemma claims that every representation of E is reducible to one defined by

$$\begin{aligned} H(u, p_w) = \mathrm{rec}(&i, p_x, j, p_y, k, p_z) \\ &[O(u, p_x, p_y, p_w), F(i, p_x, p_w, p_z, p_y), \\ &F(j, p_y, p_y, p_w, p_z), O(k, p_w, p_w, p_z, p_y)], \end{aligned} \tag{53}$$

where O and F are representations of o and f respectively. We give the

proof for this one simple example oly, since it illustrates all the points that come up in the general case.

CLAIM. *Every representation of E in* (52) *satisfies*

$$G(u, p_w) = \mathtt{rec}(i, p_x, j, p_y, k, p_z) \times [O_1(u, p_x, p_x, p_y, p_w), F_1(i, p_x, p_w, p_z, p_y), F_2(j, p_y, p_y, p_w, p_z), O_2(k, p_w, p_w, p_x, p_y)], \quad (54)$$

with suitably chosen representations O_1, O_2 *of o and* F_1, F_2 *of f.*

Proof of Claim. Let

$$o_1(x, y, z, w) = o(x, x, y, w),$$
$$f_1(x, y, z, w) = f(x, w, z, y),$$
$$f_2(x, y, z, w) = f(y, y, w, z),$$
$$o_2(x, y, z, w) = o(w, w, x, y),$$

so that

$$E = \mathtt{rec}(x, y, z)[o_1(x, y, z, w), f_1(x, y, z, w), f_2(x, y, z, w), o_2(x, y, z, w)],$$

and by 10.2 we know that every representation of E satisfies

$$G(u, p_w) = \mathtt{rec}(i, p_x, j, p_y, k, p_z) [O'_1(u, p_x, p_x, p_z, p_w), F'_1(i, p_x, p_y, p_z, p_w), F'_2(j, p_x, p_y, p_z, p_w), O'_2(k, p_x, p_y, p_z, p_w)],$$

where O'_1, etc., are representations of o_1, etc. By the way we defined composition of ipf's, there exist a representation $O''_1(u, p_x, p_y, p_z, p_w)$ of o and functions π_i, $i = 1, ..., 4$, such that

$$O'_1(u, p_x, p_y, p_z, p_w) = O''_1(u, p_x^{\pi_1}, p_x^{\pi_2}, p_y^{\pi_3}, p_w^{\pi_4}) = O_1(u, p_x, p_x, p_y, p_w),$$

where O_1 is the representation of o defined by

$$O_1(u, p_x, p_y, p_z, p_w) = O''_1(u, p_x^{\pi_1}, p_y^{\pi_2}, p_z^{\pi_3}, p_w^{\pi_4}).$$

Proof of the claim is completed by repeating the same argument for O_2, F_1, F_2.

To finish the proof of the lemma we need to find representations O and F such that G is (54) is reducible to H in (53). Set

$$O(\langle 1, u\rangle, p_x, p_y, p_z, p_w) = O_1(u, p_x^{\langle 1,\cdot\rangle}, p_y^{\langle 1,\cdot\rangle}, p_z^{\langle 2,\cdot\rangle}, p_w),$$
$$O(\langle 2, k\rangle, p_x, p_y, p_z, p_w) = O_2(k, p_x, p_y, p_z^{\langle 1,\cdot\rangle}, p_w^{\langle 2,\cdot\rangle}),$$
$$F(\langle 1, i\rangle, p_x, p_y, p_z, p_w) = F_1(i, p_x^{\langle 1,\cdot\rangle}, p_y, p_z^{\langle 2,\cdot\rangle}, p_w^{\langle 2,\cdot\rangle}),$$
$$F(\langle 2, j\rangle, p_x, p_y, p_z, p_w) = F_2(j, p_x^{\langle 2,\cdot\rangle}, p^{\langle 2,\cdot\rangle}, p_z, p_w^{\langle 2,\cdot\rangle}),$$

in each case interpreting the second clause in the definitions of O and F as the "otherwise clause," i.e.,

$$l \neq \langle 1, (l)_1\rangle \Rightarrow O(l, p_x, p_y, p_z, p_w) = O(\langle 2, (l)_1\rangle, p_x, p_y, p_z, p_w)$$

and similarly for F. Clearly O and F are representations of o and f, respectively. We prove that if H is defined by (53) with this O, F and if G is defined by (54), then

$$G(u, p_w) = H(\langle 1, u\rangle, p_w), \tag{55}$$

so that G is reducible to H and the proof of the Lemma is complete.

Fix p_w (which remains constant throughout the argument) and consider the least-fixed-point equations which determine the values of G and H:

$$\left.\begin{aligned} p_x(i) &= F_2(i, p_x, p_w, p_z, p_y) \\ p_y(j) &= F_2(j, p_y, p_y, p_w, p_z \\ p_z(k) &= O_2(k, p_w, p_w, p_x, p_y) \end{aligned}\right\} : \overline{p_x}, \overline{p_y}, \overline{p_z} \tag{56}$$

$$G(u, p_w) = O_1(u, \overline{p_x}, \overline{p_x}, \overline{p_y}, p_w)$$

$$\left.\begin{aligned} p_x(i) &= F(i, p_x, p_w, p_z, p_y) \\ p_y(j) &= F(j, p_y, p_y, p_w, p_z) \\ p_z(k) &= O(k, p_w, p_w, p_x, p_y) \end{aligned}\right\} : \tilde{p}_x, \tilde{p}_y, \tilde{p}_z \tag{57}$$

$$H(u, p_w) = O(u, \tilde{p}_x, \tilde{p}_x, \tilde{p}_y, p_w).$$

Using the fact that $\tilde{p}_x, \tilde{p}_y, \tilde{p}_z$ satisfy (57), we can verify by a direct computation that *the functions* $\tilde{p}_x(\langle 1, \cdot\rangle)$, $\tilde{p}_y(\langle 2, \cdot\rangle)$, $\tilde{p}_z(\langle 2, \cdot\rangle)$ *satisfy the system* (56), and hence

$$\overline{p_x} \leqslant \tilde{p}_x^{\langle 1,\cdot\rangle}, \qquad \overline{p_y} \leqslant \tilde{p}_y^{\langle 2,\cdot\rangle}, \qquad \overline{p_z} \leqslant \tilde{p}_z^{\langle 2,\cdot\rangle}. \tag{58}$$

To verify the converses of these inequalities we will show by an induction

on the stage n of the mutual recursion which determines the least fixed points of (57) that

$$\tilde{p}_x^{(n)}(\langle 1, \cdot \rangle) \leqslant \overline{p_x}, \qquad \tilde{p}_y^{(n)}(\langle 2, \cdot \rangle) \leqslant \tilde{p}_y, \qquad \tilde{p}_z^{(n)}(\langle 2, \cdot \rangle) \leqslant \overline{p_x}. \quad (59)$$

Looking at the induction step of the first of these, as an example,

$$\begin{aligned}\tilde{p}_x^{(n+1)}(\langle 1, i \rangle) &= F(\langle 1, i \rangle, \tilde{p}_x^{(n)}, p_w, \tilde{p}_z^{(n)}, \tilde{p}_y^{(n)}) \\ &= F_1(i, \tilde{p}_x^{(n)}(\langle 1, \cdot \rangle), p_w, \tilde{p}_z^{(n)}(\langle 2, \cdot \rangle), \tilde{p}_y^{(n)}(\langle 2, \cdot \rangle)) \\ &\leqslant F_1(i, \overline{p_x}, p_w, \overline{p_z}, \overline{p_y}) \\ &= \overline{p_x}(i),\end{aligned}$$

where the crucial $\leqslant$ step uses the induction hypothesis and the monotonicity of F_1. The computations involving $\overline{p_y}$ and $\overline{p_z}$ are similar. Putting (58) and the limit case of (59) together, we get

$$\overline{p_x} = \tilde{p}_x^{\langle 1, \cdot \rangle}, \qquad \overline{p_y} = \tilde{p}_y^{\langle 2, \cdot \rangle}, \qquad \overline{p_z} = \tilde{p}_z^{\langle 2, \cdot \rangle},$$

and hence

$$\begin{aligned}G(u, p_w) &= O_1(u, \overline{p_x}, \overline{p_x}, \overline{p_y}, p_w) \\ &= O_1(u, \tilde{p}_x^{\langle 1, \cdot \rangle}, \tilde{p}_x^{\langle 1, \cdot \rangle}, \tilde{p}_y^{\langle 2, \cdot \rangle}, p_w) \\ &= O(\langle 1, u \rangle, \tilde{p}_x, \tilde{p}_x, \tilde{p}_y, p_w) \\ &= H(\langle 1, u \rangle, p_w),\end{aligned}$$

and the proof of (55) and the lemma is complete. ∎

10.9. *Proof of the Transfer Principle* 9.1. Because of 10.7, it is enough to prove that if E, M are behavior equivalent expressions in simplified form, then every representation of E is a fixed process structure $\mathscr{A} = (\text{States}, \iota, \text{Acts}, skip, \mathscr{F})$ is also a representation of M in $\mathscr{A}$. We will show this by defining *for each given representation of* E a behavior structure $\mathscr{A}^l$ and an interpretation of $\mathscr{A}$ into $\mathscr{A}^l$, and then using the equivalence of E and M on $\mathscr{A}^l$ to get the result.

Set first

$$\begin{aligned}\text{States}^l &= N \times \text{States, with } \iota^l = (0, \iota), \\ \text{Acts}^l &= N \times \text{Acts, with } skip^l = (0, skip), \\ (v, s)(u, a) &= (v + u, sa),\end{aligned}$$

and check that the conditions on states and acts are satisfied. We complete

the definition of $\mathscr{A}^l$ below by defining a behavioral interpretation $\mathscr{F}^l$ of the function symbols of $\mathscr{L}$ which depends on the given representation of E.

Recall that behaviors take values of the form (a, w) with a an act, $w \in \{\partial, 1\}$, and to convert easily between behavior values in $\mathscr{A}$ and $\mathscr{A}^l$ set

$$\alpha(u, (a, w)) = ((u, a), w), \qquad \beta((u, a), w) = (a, w),$$

so that

$$\beta\alpha(u, (a, w)) = \beta((u, a), w) = (a, w).$$

With each $p: N \to \mathsf{B}$ we associate the behavior lp of $\mathscr{A}^l$ defined by

$$lp((u_0, s_0), ..., (u_n, s_n)) \simeq \alpha(0, p(u_0)(s_0, ..., s_n)),$$

and with each behavior $\sigma \in \mathsf{B}^l$ of $\mathscr{A}^l$ we associate the function $l^{-1}\sigma: N \to \mathsf{B}$ by

$$l^{-1}\sigma(u)(s_0, ..., s_n) \simeq \beta\sigma((u, s_0), (0, s_1), ..., (0, s_n)).$$

Note that $l\lambda(u)\varnothing = \varnothing$ and $l^{-1}\varnothing = \lambda(u)\varnothing$. It follows that for each $u \in N$ and states $s_0, ..., s_n$,

$$\begin{aligned} l^{-1}lp(u)(s_0, ..., s_n) &\simeq \beta lp((u, s_0), (0, s_1), ..., (0, s_n)) \\ &\simeq \beta\alpha(0, p(u)(s_0, ..., s_n)) \simeq p(u)(s_0, ..., s_n), \end{aligned}$$

so that

$$p = l^{-1}lp \qquad (p: N \to \mathsf{B})$$

and in particular *the map* $p \mapsto lp$ *is one-to-one.* Finally, with each representation F of $\mathscr{A}$—for simplicity binary—we associate the behavior function lF of $\mathscr{A}^l$ by

$$lF(\sigma, \tau)((u_0, s_0), ..., (u_n, s_n)) \simeq \alpha(0, F(u_0, l^{-1}\sigma, l^{-1}\tau)(s_0, ..., s_n));$$

computing again,

$$\begin{aligned} \beta lF(lp, lq)((u_0, s_0), ..., (u_n, s_n)) &\simeq \beta\alpha(0, F(u_0, l^{-1}lp, l^{-1}lq)(s_0, ..., s_n)) \\ &\simeq \beta\alpha(0, F(u_0, p, q)(s_0, ..., s_n)) \\ &\simeq F(u_0, p, q)(s_0, ..., s_n), \end{aligned}$$

so that in particular

$$F(u, p, q)(s_0, ..., s_n) \simeq \beta lF(lp, lq)((u, s_0), (0, s_1), ..., (0, s_n)) \qquad (60)$$

and *the map*

$$F \mapsto (\beta l)(F) = \lambda(p, q)\, \lambda(u, s_0, ..., s_n)\, \beta l F(lp, lq)((u, s_0), (0, s_1), ..., (0, s_n)) \tag{61}$$

is one-to-one.

Suppose now that $E(z)$ is an expression in simplified form and with the single free variable z, for example

$$E(z) \equiv \mathtt{rec}(x, y, w)[o(y, z), g(y, x, w), o(x, z), \bot],$$

the treatment of the general case being similar. Every representation of $E(z)$ is reducible to one of the form

$$H(u, r) = \mathtt{rec}(i, p, j, q, k, s)[O(u, q, r), G(i, q, p, s), O(j, p, r), \varnothing],$$

with O, G representations of o and g, respectively, and $\lambda(k)\varnothing$ the only representation of $\bot$. Define the interpretation $\mathscr{F}^l$ on the function symbols in $\mathscr{A}^l$ so that

$$\mathscr{F}^l(o) = lO, \qquad \mathscr{F}^l(g) = lG$$

and let H^l be the behavior denotation of E in $\mathscr{A}^l$,

$$\begin{aligned} H^l(\sigma_r) &= \textit{behavior}\ (\mathscr{A}^l, \sigma_r)\, E(\sigma_r) \\ &= \mathtt{rec}(\sigma_p, \sigma_q, \sigma_s)[lO(\sigma_q, \sigma_r), lG(\sigma_q, \sigma_p, \sigma_s), lO(\sigma_p, \sigma_r), \varnothing]. \end{aligned}$$

CLAIM. *For all $r: N \to \mathsf{B}$,*

$$(\beta l)\, H(lr) = \beta H^l(lr). \tag{62}$$

Proof of Claim. For each fixed r, we have

$$H(u, r) = O(u, \bar{q}, r), \qquad H^l(lr) = lO(\overline{\sigma_q}, lr),$$

where $\bar{q}$, $\overline{\sigma_q}$ are determined by the mutual resursions

$$\left.\begin{aligned} p(i) &= G(i, q, p, \varnothing)) \\ q(j) &= O(j, p, r) \end{aligned}\right\} : \bar{p}, \bar{q},$$

$$\left.\begin{aligned} \sigma_p &= lF(\sigma_q, \sigma_p)) \\ \sigma_q &= lO(\sigma_p, lr) \end{aligned}\right\} : \overline{\sigma_p}, \overline{\sigma_q}.$$

It will be enough to show that

$$l\bar{p} = \overline{\sigma_p}, \qquad l\bar{q} = \overline{\sigma_q},$$

since from this we can compute using (60)

$$\begin{aligned} H(u, r)(s_0, s_1, ..., s_n) &\simeq O(u, \bar{q}, r)(s_0, s_1, ..., s_n) \\ &\simeq \beta lO(l\bar{q}, lr)((u, s_0), (0, s_1), ..., (0, s_n)) \\ &\simeq \beta lO(\overline{\sigma_q}, lr)((u, s_0), (0, s_1), ..., (0, s_n)) \\ &\simeq \beta H'(lr)((u, s_0), (0, s_1), ..., (0, s_n)), \end{aligned}$$

from which (62) follows by another application of (60). The proof of (62) is by a routine induction on the stages of the recursions which define $\bar{p}, \bar{q}, \overline{\sigma_p}$ and $\overline{\sigma_q}$ and we will omit it.

Rephrasing the claim, we have shown that *if $E(\vec{z})$ is in simplified form, then every representation of $E(\vec{z})$ is reducible to some H_E satisfying*

$$(\beta l) H_E = \beta \; behavior(\mathscr{A}^l, \vec{\sigma}_r) \; E(\vec{\sigma}_r);$$

if $E(\vec{z}) \equiv_{beh} M(\vec{z})$, then of course

$$\beta \; behavior(\mathscr{A}^l, \vec{\sigma}_r) \; E(\vec{\sigma}_r) = \beta \; behavior(\mathscr{A}^l, \vec{\sigma}_r) \; M(\vec{\sigma}_r) = (\beta l) \; H_M,$$

where the last equality follows from another application of the Claim to M, and then by (61) we get

$$H_E = H_M.$$

Now 10.7 implies that H_M is a representation of M, which completes the proof. ∎

RECEIVED February 21, 1990; FINAL MANUSCRIPT RECEIVED January 23, 1991

REFERENCES

ABADI, M., LAMPORT, L., AND WOLPER, P. (1989), Realizable and unrealizable specifications of reactive systems, *in* "Proceedings of the 1989 ICALP."

BROCK, J. D. AND ACKERMAN, W. B. (1981), Scenarios: A model of non-determinate computation, *in* "Formalization of Programming Concepts," (J. Diaz and I. Ramos, Eds.), pp. 252–259, Lecture Notes in Computer Science, Vol. 107, Springer-Verlag, Berlin/New York.

BROY, M. (1986), A theory for nondeterminism, parallelism, communication, and concurrency, *Theoret. Comput. Sci.* **45**, 1–61.

DIJKSTRA, E. W. (1976), "A Discipline of Programming," Prentice-Hall, Englewood Cliffs, NJ.

HOARE, C. A. R. (1978), Communicating sequential processes, *Comm. ACM* **2**, 666–677.

KAHN, G. (1974), The semantics of a simple language for parallel processing, *in* "Information Processing 74, Proc. of the IFIP Congress 74" (J. R. Rosenfeld, Ed.), pp. 471–475, North-Holland, Amsterdam.

KELLER, R. M. (1978), Denotational models for parallel programs with indeterminate operators, *in* "Formal Descriptions of Programming Concepts" (E. J. Neuhold, Ed.), pp. 33–366, North-Holland, Amsterdam.

MILNER, R. (1979), "A Calculus of Communicating Systems," Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin/New York.

MILNER, R. (1983), Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25**, 267–310.

MOSCHOVAKIS, Y. N. (1989a), The formal language of recursion, *J. Symbolic Logic* **54**, 1216–1252.

MOSCHOVAKIS, Y. N. (1989b), A game-theoretic modeling of concurrency, *in* "Proceedings of the Fourth Annual Symposium on Logic in Computer Science," pp. 154–163, IEEE Comput. Soc. Press, New York.

MOSCHOVAKIS, Y. N. (1990), Computable processes, *in* "Proceedings of the 1990 POPL."

OLES, F. J. (1987), "Semantics for Concurrency without Powerdomains," Research Report RC 13195 (#54922) 10/13/87, IBM Research Division.

PARK, D. (1980), On the semantics of fair parallelism, *in* "Proc. Copenhagen Winter School," pp. 504–526, Lecture Notes in Computer Science, Vol. 104, Springer-Verlag, Berlin/New York.

PARK D. (1983), The "fairness" problem and nondeterministic computing networks, *in* "Foundations of Computer Science IV," pp. 133–162, Mathematisch Centrum, Amsterdam.

PLOTKIN, G. D. (1976), A powerdomain construction, *SIAM J. Comput.* **5**, 452–487.

PNUELI, A. AND ROSNER, R. (1989), On the synthesis of a reactive module, *in* "Proceedings of the 1989 ICALP."

PRATT, V. R. (1986), Modelling concurrency with partial orders, *Internat. J. Parallel Programming* **15**, 33–71.

SMYTH, M. B. (1978), Power domains, *J. Comput. System Sci.* **16**, 23–36.